SANDIA REPORT
SAND98-8224 • UC–405
Unlimited Release
Printed January 1998

Infrastructure for Distributed Enterprise Simulation

M. M. Johnson, A. S. Yoshimura, M. E. Goldsby, C. L. Janssen, and D. M. Nicol

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A05
Microfiche copy: A01

SAND98-8224
Unlimited Release
Printed January 1998

Distribution
Category UC-405

Infrastructure for Distributed Enterprise Simulation

Michael M. Johnson*, Ann S. Yoshimura, Michael E. Goldsby, and Curtis L. Janssen
Systems Studies Department
Sandia National Laboratories
Livermore, California 94550

David M. Nicol
Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755

ABSTRACT

Traditional discrete-event simulations employ an inherently sequential algorithm and are run on a single computer. However, the demands of many real-world problems exceed the capabilities of sequential simulation systems. Often the capacity of a computer's primary memory limits the size of the models that can be handled, and in some cases parallel execution on multiple processors could significantly reduce the simulation time.

This paper describes the development of an Infrastructure for Distributed Enterprise Simulation (IDES)—a large-scale portable parallel simulation framework developed to support Sandia National Laboratories' mission in stockpile stewardship. IDES is based on the Breathing-Time-Buckets synchronization protocol, and maps a message-based model of distributed computing onto an object-oriented programming model. IDES is portable across heterogeneous computing architectures, including single-processor systems, networks of workstations and multi-processor computers with shared or distributed memory. The system provides a simple and sufficient application programming interface that can be used by scientists to quickly model large-scale, complex enterprise systems. In the background and without involving the user, IDES is capable of making dynamic use of idle processing power available throughout the enterprise network.

* email: mmjohns@ca.sandia.gov

Acknowledgement

The authors acknowledge support by the Department of Energy through Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

CONTENTS

Intentionally Left Blank

1 INTRODUCTION

The use of parallel computers to execute discrete-event simulations has been a topic of research interest for nearly 20 years. Until recently, parallel computers could be found only in research labs, and application of parallel simulation technology was limited by the simple problem of lack of access. This has changed. Shared-memory multiprocessors have become a commodity product. Fast networks to link personal computers have become commodity products. It is now possible to order the pieces of a tremendously powerful distributed / parallel system *over the Internet* one day, receive and assemble it two days later.

But, while hardware to support large-scale simulations is readily accessible, software (typically) lags behind. In the enterprise computing world a number of tools, languages, and standards exist, e.g., Java and its development environments, CORBA and its implementations. However, systems to support large-scale distributed simulations are absent.

IDES, an Infrastructure for Distributed Enterprise Simulation, is a parallel simulation framework for complex, large-scale enterprise simulations. IDES was developed to support the study of issues of importance to national security. Many of these issues involve the analysis of complex systems. IDES is a policy driven simulation tool capable of performing decision directed analysis of complex system models. The goal of such analysis is to discover the emergent collective behavior of the system through the interaction of detailed individual submodel simulations—the definition of enterprise simulation.

In this paper we discuss issues that arose in the development of a parallel/distributed simulation system which was intended from the start to support a certain type of application, on a variety of commercially available platforms. We anticipate that the lessons we learned in the course of designing and building this system have application to other systems as well.

1.1 WHY PARALLEL SIMULATION

A great many modeling and simulation problems are either too large to run monolithically, or their performance on a single machine would be excruciatingly slow to support meaningful studies. Commercially available simulation systems are exclusively monolithic; while parallel systems exist in academia, they often assume homogeneous environments, or specific application domains (e.g. PCS networks). During the development of IDES, a number of existing simulation packages were investigated. The purely commercially systems, including BONeS, RESQ, G2, ModSim, and others were strictly sequential. Available research systems, including Maisie, OLPS, TWOS, Simpack as well as others, were for the most part optimistic, unsupported, and in a few cases, unavailable outside of academia.

Parallel simulation is beneficial in two distinct areas. First, a parallel simulation is only advantageous to time-performance when many computationally intense activities occur simultaneously in simulated time, allowing computational overhead to be executed in parallel in real-time. Second, while parallelism may not improve run-time performance for all small models, it may be necessary to support the exorbitant resource demands (virtual memory, I/O bandwidth, etc.) of large models.

The primary goal of the IDES research was to develop an object-oriented simulation system capable of supporting massive model, parallel discrete event simulations transparently across heterogeneous platforms. IDES provides a simple and sufficient Application Programming

Interface (API) which can be used to quickly model large-scale, complex systems. Relying on a common infrastructure, the system supports both distributed real-time simulation of Enterprise Models (EM), and reliable execution of Batch-Oriented Simulation (BOS).

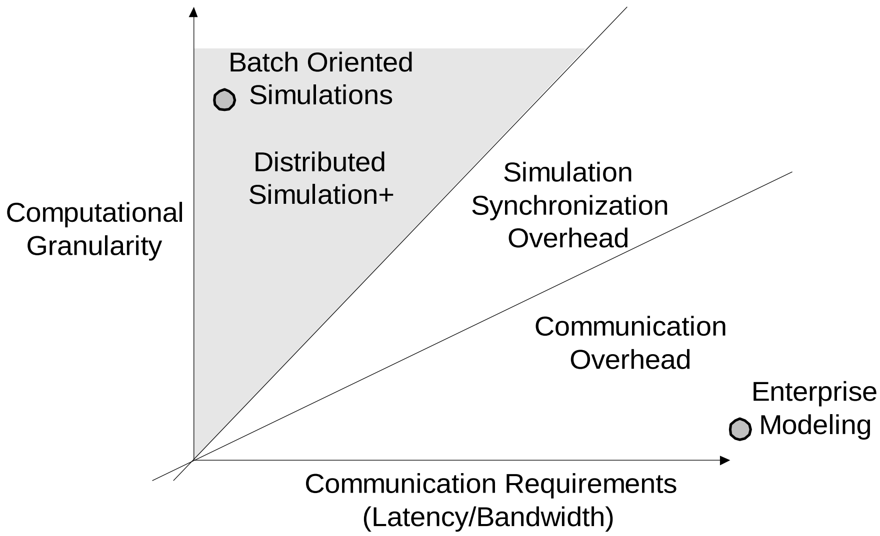

Figure 1.1: IDES simulation support.

From the beginning, the IDES system architecture has been structured to support two disparate areas of simulation: (1) Batch Oriented Simulations (BOS), where legacy codes can be bundled with input and run remotely over any number of network available machines; and (2) real-time simulation of Enterprise Models (EM), supporting object-oriented discrete-event simulation across networks of heterogeneous machines. The relationship between these two areas of supported simulation is depicted in Figure 1.1.

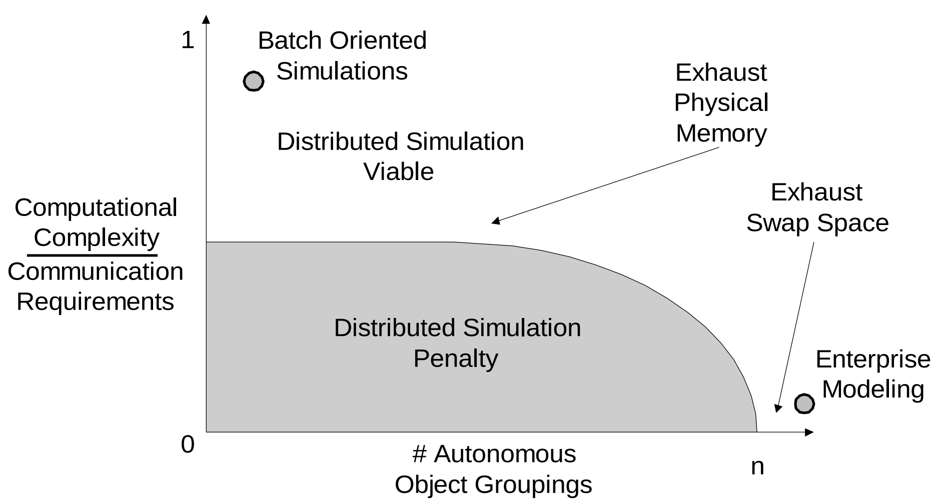

Figure 1.2: Supported areas of distributed simulation: BOS and EM.

BOS type applications are best represented by computational codes that can be easily divided into independent submodels, and for which the computational requirements of submodels is high in relation to their intercommunication requirements. BOS support provides a basis for integrating dissimilar simulation systems into a common framework. EM simulations, on the other hand, represent the parallelization of large, monolithic models that require real-time and continuous intercommunication between submodels. In general, EM simulations are extremely fine grained in comparison to BOS applications. Figure 1.2 provides another illustration of the two application areas addressed by our research. Since both of these domains of simulation require a common infrastructure—distribution of submodels, support for both on and off machine communication, and remote control of submodel invocation—it made sense to consider them as a unified design challenge. Sections 2 and 3 deal with the EM simulation aspects of IDES, our primary research focus, and section 4 details the BOS implementation.

Perhaps the most important point we wish to convey is that *capability* is our main concern, not run-time performance. Of course, execution time is a consideration, but we view it as a constraint rather than an objective function. In the enterprise computing world issues of portability, maintainability, and conformance to standards are as important as fast run-time, so much so that it is acceptable to sacrifice execution speed to provide these other capabilities.

2 IDES Performance Modeling

There are a large number of factors that potentially affect performance of the IDES system. We thought it prudent, prior to building IDES, to anticipate some of the performance considerations, by first building an analytic model of IDES to study *its* behavior.

The model recognizes that the key elements governing a submodel's behavior with respect to synchronization are, (1) its time of next event, and (2) its minimum known receive time on generated messages. A submodel's state is described by a pair of real numbers, recording these two elements. Stochastic assumptions are made about changes in those two elements as events are processed. A submodel reaches its local event horizon when its time-of-next event component dominates its receive-time component. One such model is advanced for every submodel in the system; additional assumptions about communication delay and construction of reduction trees model the inclusion of a preemptive min-reduction calculation. The end result of the model is a probability distribution of the time required to execute one BTB window. Solution of the model is computational rather than closed form.

In order to include further detail (and temporarily avoid the effort of building a numerically stable solver), we developed a simulation of this model. Performance studies using the simulation revealed the sensitivity of performance to the delay through a network interface that is shared by all processors in an SMP. This result has immediate bearing on the issue of hardware acquisition—ironically, the systems most prone to having the network interface be a performance bottleneck are the high-end larger scale (and more costly) SMP servers. Actual studies are needed to assess whether the advantage of local communication between submodels in the same SMP is enjoyed. Another point of interest was that perfect load balance is difficult if not impossible to achieve when the workload is stochastically driven. The inherent variance in the workload behavior induces a certain level of imbalance. An important conclusion to draw from this study is that complex load-balancing schemes are unlikely to be significantly more effective than simple schemes—a conclusion that has obvious bearing on IDES system design. A final lesson we learned from the simulation study was that a performance optimization we considered with regards to handling communication was usually quite effective, and hence was included in the IDES system.

One of the goals for IDES was portability across heterogeneous computing architectures, including single-processor systems, networks of workstations, and multiprocessor computers with shared or distributed memory. Given the large space of architectures across which IDES operates, we were driven to investigate the impact of different architectural features on potential designs, prior to actual implementation. A number of factors that could impact performance were considered, especially those involving synchronization, communication, and load balance. In this section we study a simple model of the synchronization strategy, and various methods that were employed to manage synchronization and communication. The model we developed reflects the effects of load imbalance, and as such may later form the basis of cost/benefit analysis of dynamic load balancing strategies.

Parallel simulation of IDES models is attractive principally because of the large memory available on distributed and parallel platforms; we anticipate massively many simulation objects, whose events will require little computation. While speedup is of course desirable, having sufficient memory is our biggest concern. This forces us towards a synchronization strategy that effectively accommodates aggregation of objects. Because of the memory constraint, our original goal was to avoid optimistic methods and their state-saving requirements by using the conservative synchronization method YAWNS (Nicol 1992, 1993). We were attracted to YAWNS by its mathematical guarantee of ample parallelism for models containing a certain type of lookahead. However, deeper analysis of IDES model characteristics showed that some objects lack the predictive capability required by YAWNS. The "optimistic" version of YAWNS is Steinman's Breathing Time Buckets (BTB) protocol, used in SPEEDES (Steinman 1992).

This section reports on our work developing a model of BTB, and an algorithm for accelerating its window computation. We are using this model to study the performance ramifications of differing architectural configurations, of differing communication costs, and to investigate the potential for alternative communication and synchronization schemes. While the model is suitable for numerical solution, at this time we are using discrete-event simulation to evaluate it.

Our work has several specific contributions. First, our model of BTB is unique in that it is tractable, it captures on-processor aggregation of objects, and captures details of run-time behavior that previous models did not. Second, we developed the "preemptive min-reduction" algorithm to accelerate detection of the BTB synchronization window edge. Third, we develop an alternative communication strategy for BTB that attempts to better utilize the communication network. Fourth, we study the projected performance of BTB on large-scale models, across various architectures, communication strategies, problem sizes, and load distributions. We found that good performance may be expected on problems of the size we anticipate in IDES, that the new communication strategy offers significant performance advantages provided that the communication interface is not overwhelmed by the offered traffic, and (surprisingly) that performance may be insensitive to moderate deviations from "perfect" load balancing. But, the largest contribution of this work is that it helped us to design IDES with some hint of the performance it would deliver and the issues that were most important in achieving good performance.

2.1 THE PROCESSOR EVENT HORIZON

Our conceptual model of the parallel simulation is that each of P processors is responsible for the simulation of a number of objects. While the processors may be organized in a shared-memory machine, a networked cluster of shared-memory machines, or a distributed-memory machine, we do assume that objects are bound together logically to be managed by a common thread of control. As the result of an object being simulated, it may produce one or more messages for other objects. Following customary PDES practice, each message is considered to have a "send-time" and a "receive-time"; the send-time is the time at which the sending object generates the message, the receive-time is the time at which that message affects the state of the recipient. A positive difference between the two reflects tentative foreknowledge of future behavior. At any instant t in simulation time, the *event horizon* $H(t)$ is the minimum receive-time at least as large as t, among all messages whose send-time is greater than t. The importance of this notion is that if $H(t) > t$, all simulation events with time-stamps between t and $H(t)$ may be safely processed without further synchronization.

Each BTB window starts with all objects synchronized at a simulation time t. The event horizon $H(t)$ is determined by allowing each object to compute optimistically forward, internally buffering all messages it generates, until either its own time-of-next-event is as large as its local event horizon (LEH)—the minimum receive-time among all messages it has generated in this window—or it learns that some other processor's LEH is smaller than its own simulation clock. At this point it participates in a computation that determines $H(t)$ as the minimum reached LEH among all objects. With $H(t)$ known, all messages with send-times less than $H(t)$ are released, as these are now known to be correct. At this step an object may receive a message with a receive-time smaller than the time of the last event the object executed. But, since the object employed state-saving during its execution, it is able to rollback to the time of that message. Also note that by construction the receive time of that straggler is at least $H(t)$, and that any message it generated with a send-time of at least $H(t)$ was withheld. Consequently the rollback does not involve sending anti-messages between objects.

The event horizon is extended by defining it in terms of send and receive times of inter-processor messages, rather than inter-object messages. As an object simulates forward it may deliver messages to other objects resident on the same processor. With this definition, the synchronization window is at least as large as before; if there is substantial messaging traffic between co-resident objects the window may be much larger. As the global synchronization that establishes the event horizon is expensive, increasing the window size serves to amortize that cost over more events executed in that window. Anti-messages need not be used if, after $H(t)$ is determined, all objects on all processors are rolled back to time $H(t)$ by means of restoring state at time $H(t)$. However, anti-messages can be used to fine-tune the rollbacks to bring back only those objects that must be rolled back, and to bring them only as far back in time as the messaging behavior warrants.

Two previous analytic efforts shed light on BTB performance, but neither attempted to capture the behavior of the protocol in real time. The first is a stochastic study of YAWNS (Nicol 1991, 1993), that focused on the density of events within a synchronization window. The analysis of BTB in (Steinman 1994) used differential equations, but came to qualitatively the same conclusion—the density of concurrently simulatable events in a window increases as the number of objects increases, even though the window size is decreasing.

Performance modeling of parallel discrete-event simulations has proven to be a rich area, with a number of models addressing high level aspects of different synchronization protocols, e.g., (Nicol 1991, Felderman & Kleinrock 1991, Ferscha 1995, Gupta et.al. 1996). Our work is unique in this context with its focus on exploring performance on different architectures, using different strategies for communication.

2.2 A Model of a Parallel Simulation

We assume that objects are mapped to processors with a mapping that does not change within a synchronization window. We let n_i denote the number of objects assigned to processor i. Of these, b_i are *boundary objects*, capable of generating messages destined for objects on other processors. The remaining $x_i = n_i - b_i$ objects are *interior*, as they communicate only with objects assigned to their own processor. Obviously, the classification of any given object as boundary or interior is driven by the mapping of objects to processors; it is not a classification intrinsic to the simulation. For simplicity we assume that $\min\{x_i, b_i\}$ divides $\max\{x_i, b_i\}$ evenly.

Each processor maintains its own event list, with events associated with all objects being placed on that list. The processor manages the event list in the usual way. We presume that the distribution of simulation time between successive events is random, with probability density function a_i on processor i. The receive-time of a message is larger than its send time by a random amount, with probability density function t_i. We assume that a boundary object generates exactly one message; it is not particularly difficult to extend this to multiple messages, in which case t_i describes the minimum time-stamp among them, but we have not pushed through all the ramifications of such an extension. We will occasionally use the complementary cumulative distribution functions

$$A_i(x) = \int_x^\infty a_i(s)ds \qquad \text{and} \qquad R_i(x) = \int_x^\infty t_i(s)ds$$

We assume that the simulation of boundary events is distributed evenly among simulation of interior events: if $b_i < x_i$, then every $(x_i/b_i + 1)^{th}$ event is for a boundary object and if $x_i < h_i$ every $(b_i/x_i + 1)^{th}$ event is for an interior object. Knowledge of step number k then completely determines whether the associated event execution is for an interior or boundary object.

We assume that execution of each event requires unit time, and that transmission of a message between any two processors requires m units of time, m being integer. Event processing in IDES is fine-grained relative to communication delays; hence our assumption that communication costs are an integer multiple of event execution costs reflects this. For the ease of exposition we assume each processor has the same unit of execution time; it is not difficult to allow different processors to have different event execution speeds.

The IDES project is portable across different communication architectures. Salient features of different communication networks IDES will run on are modeled with two attributes: clustered/unclustered and serial/parallel. An architecture has a clustered attribute if subsets of processors organized as a machine can communicate through shared memory virtually instantaneously, at least relative to communication off-machine. For such architectures each "processor" we talk about corresponds to a thread in a machine, a thread that when executing is responsible for the simulation of a static subset of objects. Despite the shared memory, we still consider objects to be assigned to processors. We consider an unclustered architecture to consist of clusters of one CPU each. Next we describe inter-machine communication as being serial (like an Ethernet) or parallel. In the latter case we presume that a machine has only one network port for all its processors. The machine may send only one message at a time, and messages from processors on the machine cannot be sent simultaneously, however machines may transmit in parallel with no contention. We tacitly assume that a machine may receive messages simultaneously. For the case of clustered processors we let p_c denote the fraction of a processor's messages that are targeted within the common machine.

Our overall approach is to develop models of execution behavior, synchronization behavior, and communication behavior. The synchronization and communication behaviors are driven as a function of the execution behavior, which in turn is described in terms of state evolution equations. These equations are exact, but will contain unrealizable entities such as ∞. Ultimately we will solve these equations approximately using numerical techniques. Those techniques will necessarily discretize the state space and will necessarily truncate infinite expansions. Although an exact closed form solution may be unrealizable, we find value in presenting the equations in exact form, and leave the task of approximation to the numerics. We will analyze the computational complexity of the numerical solution and show that it is almost linear in the number of domain points at which the solution is constructed.

2.2.1 State Evolution Model

The key idea behind the analytic model is that the execution state of a processor is a pair (s, r), where s is the time-of-next-event (time-stamp on the next event to execute), and r is the least receive-time among all messages the processor has generated since the beginning of the synchronization window. The state evolves from (s, r) to (s', r') in a single event execution if the receive-time of a communication generated at time s is r', and if then the time-of-next-event is s'. The processor terminates its window processing in state (s', r') if $s' \geq r'$.

With this simple notion of processor execution state, we can express a single processor's state-evolution equations in wallclock time for a single window, with a change in state occurring every unit of execution time. We will always express the earliest window time as time 0, even though this is true only for the first window. One window is representative of all windows so long as all processors are synchronized to begin execution at the beginning of the window. In SPEEDES this may not be the case, as objects may not have all been rolled back to the event horizon of the previous window. Our assumption that they are can only serve to worsen actual performance.

The state-evolution equations are expressed recursively in terms of the probability density functions over the state-space, at each unit of execution time. For all processors i = 0, 1,..., P - 1 and time-steps k = 1, 2, . . . let $f_i^{(k)}(s, r)$ be the joint probability density function over the space [0, ∞] × [0, ∞] of processor i's state after executing the k^{th} event in the window. We define $f_i^{(k)}(s, r)$ = 0 whenever $s > r$; $f_i^{(k)}$ should be thought of as an unconditional density over non-terminated states (we will define other density functions over terminated states also, in order to capture termination probabilities). To obtain the probability of the processor having non-terminal state in any region S after executing an event at real-time k, we integrate $f_i^{(k)}(s, r)$ over S. We allow $f_i^{(k)}(s, r)$ to contain Dirichlet "spikes" so that this formulation encompasses discrete probability masses in the state-space (as may occur if α_i and τ_i have discrete mass at some points). Initially we take $f_i^{(0)}(0, T)$ to have a spike with value 1 and $f_i^{(0)}(s, r)$ to be zero everywhere else, where T is some arbitrarily large number certain to exceed the simulation's termination time.

Supposing $f_i^{(k-1)}(s, r)$ to be defined everywhere, consider the state evolution at step k for an interior object. We define a density function $g_i^{(k)}(s, r)$ to describe the effect the k^{th} event execution has on the time-of-next-event state component (as the receive-time component cannot change without a communication):

$$g_i^{(k)}(s,r) = \int_0^{\min\{s,r\}} f_i^{(k-1)}(a,r)a_i(s-a)da \tag{1}$$

This expression reflects that to reach state (s, r) at step k, one must first be in a non-terminated state (a, r) with $a \leq \min\{s, r\}$, and from this state advance the simulation clock by precisely $s - a$ units of simulation time. Since $g_i^{(k)}$ may be non-zero at terminal states, we can extract the probability of termination at step k by integrating $g_i^{(k)}$ over the space of terminal states.

The state evolution at a step involving a communication is similar, except that it is possible for the receive-time component of the state to change:

$$h_i^{(k)}(s,r) = \int_0^{\min\{s,r\}} \int_{r^+}^{\infty} f_i^{(k-1)}(a,b) t_i(r-a) a_i(s-a)\, db\, da + \int_0^{\min\{s,r\}} f_i^{(k-1)}(a,r) R_i(r-a) a_i(s-a)\, da,$$

which can be rearranged as

$$h_i^{(k)}(s,r) = \int_0^{\min\{s,r\}} t_i(r-a) a_i(s-a) \left(\int_{r^+}^{\infty} f_i^{(k-1)}(a,b)\, db \right) da + \int_0^{\min\{s,r\}} f_i^{(k-1)}(a,r) R_i(r-a) a_i(s-a)\, da. \tag{2}$$

The first term accounts for transitions where both components of the state change. As the source and target states are known, the precise values of the two random variables necessary to effect the transition may be specified. The notation r^+ reminds us that the source states over which we integrate must have a receive-time component strictly greater than r; the integration specifically excludes states (a, r). The second term accounts for transitions from states where the receive-time component is already r; for this the value of the receive-time of the communication generated must be at least r.

It is notationally convenient to use a density function $d_i^{(k)}$ that "switches" between $g_i^{(k)}$ and $h_i^{(k)}$ as a function of k: for k involving interior objects, $d_i^{(k)} = g_i^{(k)}$, and for k involving boundary objects $d_i^{(k)} = h_i^{(k)}$. The probability density function $f_i^{(k)}$ for non-terminal states after step k can be formulated from these definitions. We define $f_i^{(k)}(s, r) = 0$ for all k and all states (s, r) where $s \geq r$. For non-terminal states (s, r) we define $f_i^{(k)}(s, r) = d_i^{(k)}(s, r)$.

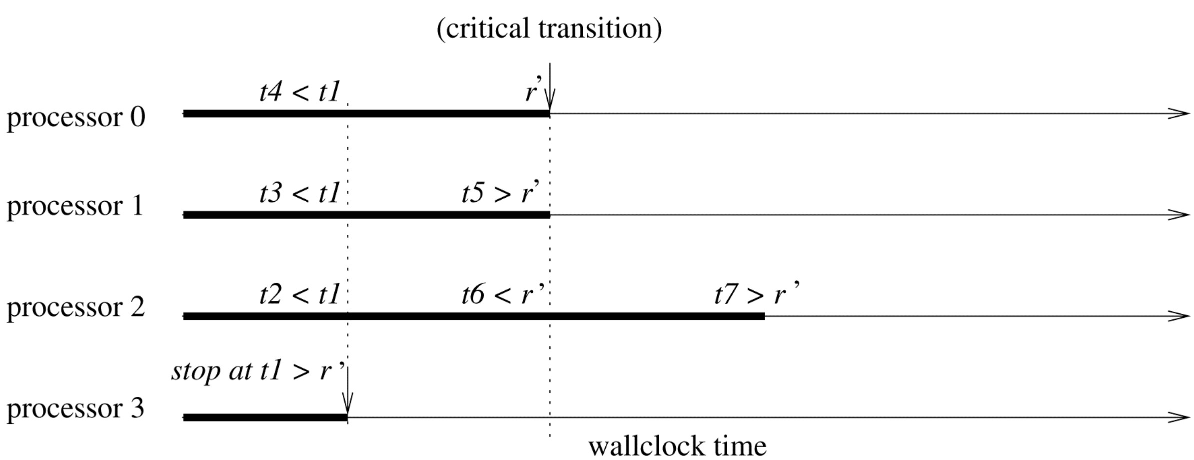

Figure 2.1: Discovery of the processor event horizon.

2.2.2 Window Execution Time

Next we consider how to use the density functions to describe the probability distribution of the time required to simulate a window, if communication is instantaneous. Later subsections will include the cost of more realistic synchronization, and communication.

We suppose for a moment that whenever a processor reaches its local event horizon r, that value is known immediately to all processors. Any processor that has not yet reached its local event horizon can stop immediately if its own simulation clock exceeds r; likewise, a processor can stop once its time-of-next-event is larger than the smallest known local event time, even if that processor has not yet reached its own local event horizon.

We say that the *critical transition* occurs at (i, n, r') if processor i reaches its local event horizon r' after step n, and the global event horizon ends up being r'. The fact that the transition is critical may not be immediately evident to the simulation. Observe that processor i need not be the first to terminate, nor will the window expansion necessarily stop instantly. In the former case another processor can reach a larger local event horizon in less time, in the latter case there may be an unterminated processor whose time-of-next-event is less than r'. These points are illustrated in Figure 2.1. Processor 3 is first to terminate the window, with a local event horizon of $t1$. At that instant, the time-of-next-event of every other processor is less than $t1$, and so all other processors continue. Later processor 0 terminates with a local event horizon of r'. At this instant processor 1's time-of-next-event is $t5 > r'$, so processor 1 stops despite not having reached its local event horizon. Processor 2's time-of-next-event is only $t6 < r'$ though, and so processor 2 continues. Later, when processor 2's time of next event is $t7 > r'$, it stops and the window is finally terminated.

There is a probability density function $G(i, n, r')$ associated with the critical transition. Knowledge of this function's structure will allow us to compute many other performance measures, for by conditioning on (i, n, r') the description of the rest of the system is simplified. $G(i, n, r')$ is the product of certain probabilities, and a density function. First observe that if processor i is to define the global event horizon at simulation time r', then none of the other processors can have local event horizons less than r'. The probability $L_j(r')$ that processor j's local event horizon is at least as large as r' is easy to express as the sum over all steps k of entering a terminal state (s, t) with $t \geq r'$. This simply means integrating $d_j^{(k)}(s, t)$ over all (s, t) with $r' \leq s$ and $t \leq s$.

$$L_j(r') = \sum_{k=1}^{\infty} \int_{r'}^{\infty} \int_{0}^{s} d_j^{(k)}(s,t)\,dt\,ds \tag{3}$$

The probability that all processors other than i have local event horizons as large as r' is just the product of their values $L_j(r')$. We multiply this product with a density function that expresses how processor i may reach local event horizon r' at step n. Obviously, the construction of this density function must consider that processor i is constrained from tracing a path that terminates in a step prior to n, or at a local event horizon other than r'. Note that if processor i is in a non-terminal state (a, b) with $r' \leq b$ after step $n - 1$, then whatever path it took to reach (a, b) satisfies this constraint; density $f_i^{(n-1)}(a, b)$ already reflects the constraint. Furthermore, the only transition path to a terminated state (s, r') after step n is from some non-terminated state (a, b) after step $n - 1$, with $r' \leq b$; e.g., every path to state (s, r') satisfies the conditioning constraint, so that no further adjustment need be applied to its density function. This realization completes what is needed for the definition of density $G(i, n, r')$:

$$G(i,n,r') = \left(\prod_{j \neq i} L_j(r') \right) \int_{r'}^{\infty} d_i^{(n)}(a,r')da$$

By conditioning on critical transition parameters (i, n, r'), we can compute the distribution of the remaining number of execution steps that another processor takes before terminating. The conditioning alters its density equations at each step, scaling them by one over the probability of the conditioning constraint occurring naturally in the distribution. For $j \neq i$ we denote processor j's conditional density by $\boldsymbol{f}_j^{(k)}(a, b, r')$.

First observe that over the region $A_j(r') = \{(a, b) \mid a < b \text{ and } b > r\}$ (i.e. non-terminated states that do not violate the conditioning), $\boldsymbol{f}_j^{(k)}$ is proportional to $f_j^{(k)}$, e.g., $\boldsymbol{f}_j^{(k)}(a, b) = b_j^{(k)}(r')\, f_j^{(k)}(a, b, r')$ for some $b_j^{(k)}(r')$. This can be proven by induction on k, it is a straightforward consequence of the fact that the density function f appears as a single term in every integral in equations (1) and (2). This fact has a useful application. In the unconditional system of equations, let $g_j^{(k)}(r')$ be the fraction of probability mass that transfers at step k from $A_j(r')$ to region $B_j(r') = \{(a, b) \mid b < r'\}$, e.g.

$$g_j^{(k)}(r') = \frac{\int_0^{r'} \int_{r'}^{\infty} f_j^{(k-1)}(a,b)(1 - R(r'-a))dbda}{\int_{A(r')} f_j^{(k-1)}(a,b)dbda} \tag{4}$$

If we were to condition on processor j not entering region $B_j(r')$ during the first $k - 1$ steps and then allow any transition at step k, the fraction of conditional probability mass over $A(r')$ that transfers to region $B(r')$ is precisely $g_j^{(k)}(r')$, again because $\boldsymbol{f}_j$ is proportional to f_j over $A(r')$.

We express $\boldsymbol{f}_j^{(k)}$ recursively, beginning with $\boldsymbol{f}_j^{(0)} = f_j^{(0)}$. Given $\boldsymbol{f}_j^{(k-1)}$ for $k \geq 1$, if step k corresponds to an interior object we define $\boldsymbol{g}_j^{(k)}$ over the entire domain by replacing $f_j^{(k-1)}$ in equation (1) with $\boldsymbol{f}_j^{(k-1)}$; we define $\boldsymbol{h}_j^{(k)}$ with a similar substitution in equation (2) if step k corresponds to a boundary object. Then we define $\boldsymbol{d}_j^{(k)}$ to switch between $\boldsymbol{g}_j^{(k)}$ and $\boldsymbol{h}_j^{(k)}$ as a function of k. Function $\boldsymbol{d}_j^{(k)}$ is defined over the entire domain; it expresses how the conditional probability mass after step $k - 1$ would spread after step k if no constraints were placed on the step k transition. But, we must condition on step k not entering forbidden territory, and do so by scaling the value of $\boldsymbol{d}_j^{(k)}$ by the probability of not transitioning into the forbidden region. This probability is just one minus $g_j^{(k)}(r')$, which gives us

$$\boldsymbol{f}_j^{(k)}(a,b,r') = \begin{cases} \boldsymbol{d}_j^{(k)}(a,b,r')/(1 - g_j^{(k)}(r')) & \textit{for}\,(a,b) \in A(r') \\ 0 & \textit{otherwise} \end{cases}$$

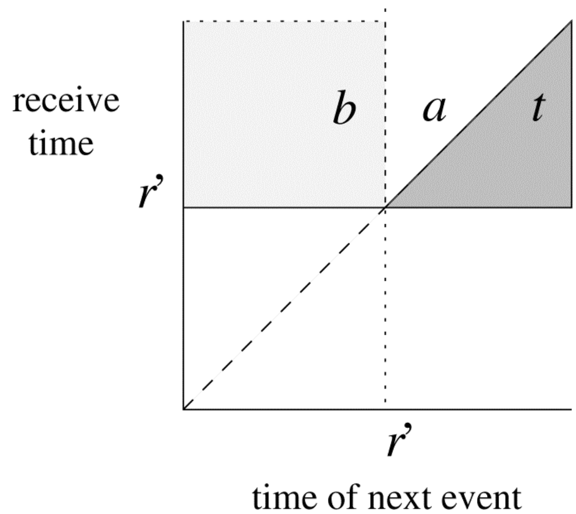

Figure 2.2: Three feasible regions of the state space for processor j, conditioned on critical transition (i, n, r').

This expression reveals the constant of proportionality between $\boldsymbol{f}_j^{(k)}$ and $f_j^{(k)}$; it is a straightforward exercise in induction to prove that

$$\boldsymbol{f}_j^{(k)}(a,b,r') = \begin{cases} f_j^{(k)}(a,b) / \prod_{z=1}^{k} (1 - g_j^{(z)}(r')) & \textit{for } (a,b) \in A(r') \\ 0 & \textit{otherwise} \end{cases}$$

After step k, a processor that was not terminated by step $k - 1$ will be in one of three regions of the state space, illustrated by Figure 2.2. The horizontal axis gives the time-of-next-event state component, the vertical axis the receive-time component. Terminated states lie below the diagonal. Region b ("before" time r') is comprised of non-terminated states whose time-of-next-event component is still less than r'; region a ("after" time r') has non-terminated states whose time-of-next-event component is as large as r'; region t has states reflecting termination in states with both components as large as r'. Then for each step k we define $S_{j,b}^{(k)}(r')$, $S_{j,a}^{(k)}(r')$, and $S_{j,t}^{(k)}(r')$ to be the integral of $f_j^{(k)}$ over regions b, a, and t respectively; we then define $S_{j,T}^{(k)}(r') = S_{j,a}^{(k)}(r') + S_{j,b}^{(k)}(r') + S_{j,t}^{(k)}(r')$. These definitions give us the values needed to describe the probabilistic behavior of processors with respect to the critical transition.

Let $T_j(n, r')$ denote the random number of remaining steps (after step n) that processor j takes until stopping. One possibility is that j terminated before or at step n; another is if after step n its time-of-next-event is larger than r'. These observations give

$$\Pr\{?_j(n,r') = 0\} = \left(\sum_{k=1}^{n} S_{j,t}^{(k)}(r') \right) + S_{j,a}^{(n)}(r') \qquad (5)$$

For processor j to continue another $y > 0$ steps it must be in region "b" for $n + y - 1$ successive steps, and then pass into region "a" or "t":

$$\Pr\{?_j(n,r')=y\}=\left(\prod_{z=1}^{n+y-1}\frac{S_{j,b}^{(z)}(r')}{S_{j,T}^{(z)}(r')}\right)\left(\frac{S_{j,a}^{(n+y)}(r')+S_{j,t}^{(n+y)}(r')}{S_{j,\mathrm{T}}^{(n+y)}(r')}\right) \qquad (6)$$

Each term of the first product is the probability of being in region *b* after step *z*, given that the processor did not terminate before step *z*.

The BTB window terminates once the last processor stops, e.g. at time

$$n+\max_{j\neq i}\{?_j(n,r')\}. \qquad (7)$$

Because arguments of the max are independent, that distribution is expressed using standard methods of order statistics and the distribution function $W(i, n, r')$ of the termination time simply adds n. The unconditional distribution of the time to complete a window can thus be viewed as being instances of distributions $W(i, n, r')$ "mixed" by the density function $G(i, n, r')$, e.g.

$$W=\sum_{i=0}^{P-1}\int_0^\infty G(i,n,r)W(i,n,r)dr. \qquad (8)$$

2.2.3 End Of Window Calculation

The end-of-window calculation is critical to BTB's performance. A naive and inefficient means of computing the global event horizon would be to use a min-reduction on each processor's local event horizon; the entire computation would wait for the last processor to reach its local event horizon. A better way, as modeled in the previous section, is to disseminate local event horizon times as they are reached, and rein in processors whose simulation clocks have already advanced beyond the global event horizon. This can be accomplished with a device we call a preemptive min-reduction. Implementation of a preemptive min-reduction depends on whether the communication architecture is parallel or serial. In the parallel case we use the usual reduction-tree organization for calculating an associative reduction in $\lceil \log P \rceil$ steps. In such an approach, once a processor has a value to reduce, it goes through a sequence of pairwise synchronizations with a selected group of $\lceil \log P \rceil$ other processors. At each synchronization the processors exchange their "working minimums", and both retain the minimum of the exchanged values as the new working minimum. After the last step every processor's working minimum is the global minimum.

The preemptive min-reduction requires more asynchrony between itself and application code than an ordinary reduction. A processor must frequently check for synchronization messages, and must maintain a working minimum of the smallest value sent to it by any synchronization neighbor. Before executing an event, the processor compares the time-stamp with the working minimum, possibly stopping as a result. If it does stop, it immediately engages in the min-reduction synchronization logic, offering the working minimum as its value. If a processor reaches its local event horizon before being preemptively stopped, it offers its own local event horizon to the min-reduction logic. The *only* way the preemptive min-reduction differs from an ordinary reduction is that a processor may enter the reduction logic before it ordinarily would, passing as its own value one that it was sent by a synchronization partner.

The preemptive min-reduction slows down the simulation by a limited amount over the ideal case when communication is instantaneous. Stated precisely, if under instantaneous communication t_{ideal} execution steps are needed to completely terminate a window, then the *additional* time required when communication is not instantaneous is no more than $2m\lceil \log P' \rceil$ (provided that there is no contention in the communication channel) where P' is the number of machines. A proof of this claim is given in Appendix B. For our modeling purposes this is a useful result, as it permits us to put a nice upper bound on the window execution time by simply adding another cost to the time predicted under the instantaneous communication model.

The preemptive min-reduction can be implemented in a serial communication medium by modifying any distributed termination detection algorithm. Termination detection messages are augmented to carry with them the lowest known local event horizon, and processors update their working minimum as a result of receiving these messages. One considers a processor to be terminated if either it has reached its local event horizon, or if it has been stopped by receipt of a sufficiently small working minimum. Assuming a token-passing approach where the token is always in motion between processors, the end of the window will be discovered and known to all processors in no more than $2P'm$ time once the last processor has terminated. The $2P'$ term comes from standard termination detection results showing that no more than 2 round-trips are needed to detect termination, once it has occurred.

To account for the cost of detecting the window termination, we will add either $2m\lceil \log P' \rceil$ or $2mP'$ to the window execution time, as appropriate for the communication model. Since this is an upper bound, performance measures so obtained are no better than measures obtained from a more exact analysis.

2.2.4 MESSAGE TRANSFER

Under the usual definition of BTB, once a processor knows the global event horizon, it sends all messages whose send-times fall within the window. We account for the cost of communication at the sender's end; we assume that if a machine has K messages to send, then those messages are sent serially. A transfer is assumed to take zero time if the message is for a processor in the same machine, and otherwise takes time m. If the communication model is "parallel", then all machines are permitted to send messages concurrently without contention. If the communication model is "serial", only one message at a time is permitted on the communication channel.

Given critical transition parameters (i, n, r'), let $X_j(i, n, r')$ denote the random number of steps processor j requires for its time-of-next-event to be as large as r'. Recalling Figure 2.2, its distribution is given by

$$\Pr\{?_j(i,n,r') = y\} = \left(\prod_{z=1}^{y-1} \frac{S_{j,b}^{(z)}(r')}{S_{j,T}^{(z)}(r')} \right) \left(1 - \frac{S_{j,b}^{(y)}(r')}{S_{j,T}^{(y)}(r')} \right) \qquad (9)$$

If we know that $X_j(i, n, r') = y$, then we know that the number of messages processor j has to deliver is (ignoring that it should be integer) yF_j, where F_j is the relative fraction of boundary objects on processor j that communicate off-machine. Recalling that we assume only one network connection per machine, the random network load offered per machine is the convolution of independent random loads generated by the processors comprising that machine. Denoting the set of processors in machine k as M_k, and remembering that the communication phase finishes when the last message is delivered, the communication transfer cost for the case of parallel communication is

$$C_p(i,n,r') = m \max_{machines\ k}\{\sum_{j \in M_k} F_j X_j(i,n,r')\}.$$

This expression is assumed to use the fact that given critical transition parameters (i, n, r'), then $X_i(i, n, r') = F_i n$. Once the distribution of the convolutions for each machine is known, the distribution of this max is expressed using standard methods of order statistics.

In the case of serial communication the conditional communication cost is the sum

$$C_s(i,n,r') = m\left(nF_i + \sum_{j \neq i} F_j X_j(i,n,r') \right).$$

IDES aims to explore a communication strategy that does not withhold message traffic like standard BTB. Under this strategy messages are transmitted as they are generated, but it requires the recipients to filter out any messages whose send-times exceed the global event horizon. Assuming that communication can occur in parallel with computation (which is reasonable with today's architectures that dedicate processors to manage communication), this strategy eliminates the serialization of communication at the end of the window. However, the cost of the strategy is that more messages may be sent; at the end of the window there may be a backlog of irrelevant messages waiting for delivery. Our model will help to assess situations where the alternative strategy may be beneficial.

Given the threat of excessive communication, it is critically important that the global event horizon be known as soon as possible to all processors. One way to accomplish this is to have each processor broadcast its local event horizon if it should reach it before being otherwise preempted. For the case of parallel communication, standard broadcast trees will implement the necessary communication. Given any two processors i and j, we denote by $h(i, j)$ the number of communication hops between a broadcast by i and its receipt by j, and know that j will receive a message broadcast by i in $m \times h(i, j)$ time.

Processor j will terminate when it either reaches its own local event horizon, reaches another processor's local event horizon before its own, or receives a broadcasted local event horizon smaller than its own time-of-next-event. Conditioned on critical transition parameters (i, n, r'), we find an upper bound on j's termination time by assuming that it can only be preempted by processor i's broadcast. If j were allowed to run until it reached its local event horizon, the number of steps executed, denoted $L_j(r')$, is distributed as

$$\Pr\{L_j(r') = y\} = \left(\prod_{z=1}^{y-1} \frac{S_{j,b}^{(z)}(r') + S_{j,a}^{(z)}(r')}{S_{j,T}^{(z)}(r')} \right) \frac{S_{j,t}^{(y)}(r')}{S_{j,T}^{(y)}(r')}.$$

There are three cases governing j's termination. The first occurs when $L_j(r') < n + m \times h(i, j)$, in which case processor j has reached its local event horizon before being preempted by i. The second case is if $X_j(i, n, r') < n + m \times h(i, j) < L_j(r')$, where processor j is preempted by the arrival of processor i's broadcast. The final case is when $X_j(i, n, r') > n + m \times h(i, j)$, so that processor i's message preempts j some time after arrival.

The distribution of the number of execution steps $E_j(i, n, r')$ that processor j takes before terminating is a mixture (sum) of

- the probability that $L_j(r') < n + m \times h(i, j)$, times the distribution of $L_j(r')$ conditioned on this inequality;
- the probability that $X_j(i, n, r') < n + m \times h(i, j) < L_j(r')$, times the distribution of $X_j(i, n, r')$, conditioned on this inequality;
- the probability that $X_j(i, n, r') > n + m \times h(i, j)$, times the distribution of $X_j(i, n, r')$ conditioned on this inequality.

The terms expressing $E_j(i, n, r')$'s distribution are obtained mechanically.

Having described the distribution of the number of messages processor j will send, we now turn to the description of the time required to send them. The critical factor is whether the communication channel can keep up with the load placed upon it. If the average number of steps between inter-machine communications is as large as m, then no message has to wait when it is handed off for transmission, and processor j's message transfer phase is completed m time units after its last transmission. Otherwise the communication channel is busy from the time the first message is generated until the last one is delivered. These times are all deterministic functions of $E_j(i, n, r')$, whose distribution we know.

While the case of serial communication has the most to gain from the alternative messaging strategy, it is much harder to analyze. The aggregate rate of messages to be sent can be computed assuming that no processor has terminated, but the rate decreases as processors stop. We can bound performance though by assuming that the initial aggregate rate is sustained throughout the window. Then, as before, there are two cases. If the aggregate rate is less than $1/m$ messages offered per unit time, then a message does not wait, and the overall communication phase is terminated m time units after the last message is generated. Otherwise the communication channel is saturated and we compute the end of the communication phase by determining when the last message offered to the channel is finally delivered. These measures are again deterministic in the length of the computation phase, whose distribution is known. Although we have not done so here (in the interests of space), one can express these functions in a manner similar to $C_s()$ and $C_p()$, except that it will be easiest to express the sum of the window termination time plus communication delay in a single function of a processor's stopping time.

2.2.5 Overall Performance Measures

Solution of the equations describing state occupancy yields a great deal of information about the model. We have described entire probability distributions, and so can examine rare-event probabilities, correlations, maximum buffer needs, and other measures that are costly to estimate using discrete event simulation. For our immediate purpose though we consider one overall measure of performance that is directly related to the speed at which the simulation is executed.

That measure is the average ratio of execution time per window to simulation time advanced in that window. The smaller the ratio, the faster the simulation is executing. Conditioned on the critical transition parameters, the ratio is

$$
\begin{aligned}
U(i,r,r') &= \frac{E[M(i,n,r') + C_p(i,n,r') + 2m\lceil \log P \rceil]}{r'} \\
&= \frac{E[M(i,n,r') + E[C_p(i,n,r')] + 2m\lceil \log P \rceil]}{r'}
\end{aligned}
\tag{10}
$$

The unconditional average time per unit simulation is now obtained by using the density function for the critical transition

$$
U = \sum_{i=0}^{P-1} \sum_{n=1}^{\infty} \int_0^{\infty} G(i,n,r) U(i,n,r)\, dr. \tag{11}
$$

A model of this type is of limited utility if its solution takes too long to be practical. We have done a complexity analysis of the solution procedure and found that the asymptotic time complexity is $O(PKN^2 \log N)$, where P is the number of processors, K is the number of time-steps for whose densities we solve, and the two-dimensional domain at a step is approximated with N^2 points. Thus, the solution is almost linear in the total number of state points where probabilities are calculated. In Appendix C we sketch the computational complexity of solving the various equations describing our model's behavior.

2.3 EXPERIMENTS

We have written a discrete-event simulator of the abstract model described in this paper. The simulator accepts a number of parameters describing a model instance, including

- The number of objects.
- The number of machines and the number of processors per machine.
- The network interface delay (NID), in event execution time-steps.
- Whether the inter-machine network is parallel or serial.
- An initial random number seed.
- The simulation termination time.
- Parameters of τ distribution (assumed to be identical for all objects).
- Load imbalance parameter.

We derive other simulation parameters from an abstract model of an underlying simulation problem. Given N^2 simulation objects, we presume them to be arranged in an $N \times N$ torus. An object is presumed to be able to communicate with adjacent objects. The rest of the simulation parameters for a balanced workload model are determined by assuming the objects are partitioned rectilinearly, with squares mapped to machines. A machine's square is partitioned further, evenly, among processors. Given n_i objects mapped to processor i, we determine the frequency of boundary object events to interior object events (or vice-versa) by the ratio of the number objects lying along the boundary of a processor's square to those within its interior. In the limit of increasing n_i there are $4n_i^{1/2}$ interior object events for every boundary object event.

For a large number of objects, the arrival process of events will be nearly Poisson; correspondingly we take a_i to be exponentially distributed, with rate n_i (scaling simulation time so that an object's event rate is 1). The gap between send and receive time is taken to be a constant c, plus a non-negative Gaussian. Throughout our experiments we use $c = 0.1$ and a non-negative Gaussian with mean 1 and standard deviation 0.5.

We control the load imbalance parametrically, as follows. Let $f_1 \geq f_2 \geq \ldots \geq f_P$ be P fractions that sum to 1. Considering how objects are partitioned among processors, the width of the i^{th} column of processors spans f_i of the width of the domain; likewise the height of the i^{th} row of processors spans f_i of the domain. If we describe a processor in terms of its (i, j) position in the matrix of processors, that processor gets $f_i \times f_j \times N^2$ objects. This is illustrated in Figure 2.3. The simulator allows us to specify the ratio of f_1^2 to $1/P$, i.e. the ratio of maximum workload to average workload. If we further assume that $f_i - f_{i+1}$ is constant for all $i = 1,2, \ldots, P^{1/2}$, then ratio f_1^2P completely determines the load distribution.

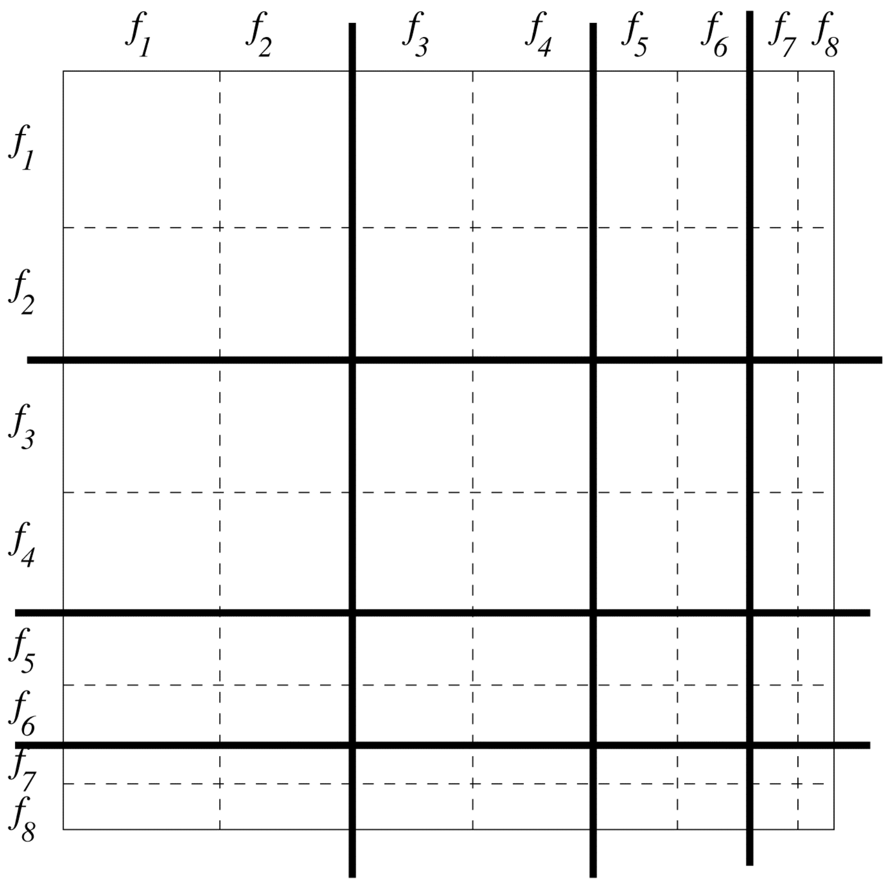

Figure 2.3: Abstract workload model partitioned among machines and processors.

In these experiments we compute "speedup" as the ratio of the total number of committed events executed, divided by the (simulated) wallclock time of the parallel simulation run. Also, the experiments are run for a long time. Informal experiments show that the statistics we observe vary by less than 1% given different initial random number seeds. Consequently we forsook the construction of independent replications and confidence intervals with good assurance that the data we observe is statistically sound.

In our first set of experiments we sought to get an initial feel for the relative importance of presending messages or not, and using parallel communication channels or serial channels. We examined a "base-line" model with 512^2 = 0.25M objects, executed on an architecture with 16 machines, each machine containing 4 processors. To be conservative we selected a network interface delay of 5; we also considered a perfectly balanced workload. A speedup of 46 was predicted for a parallel communication channel with no pre-sending; this figure increases to 55.5 when pre-sending is used. In the serial case, without pre-sending the speedup is 25, but with pre-sending it drops to 14.7.

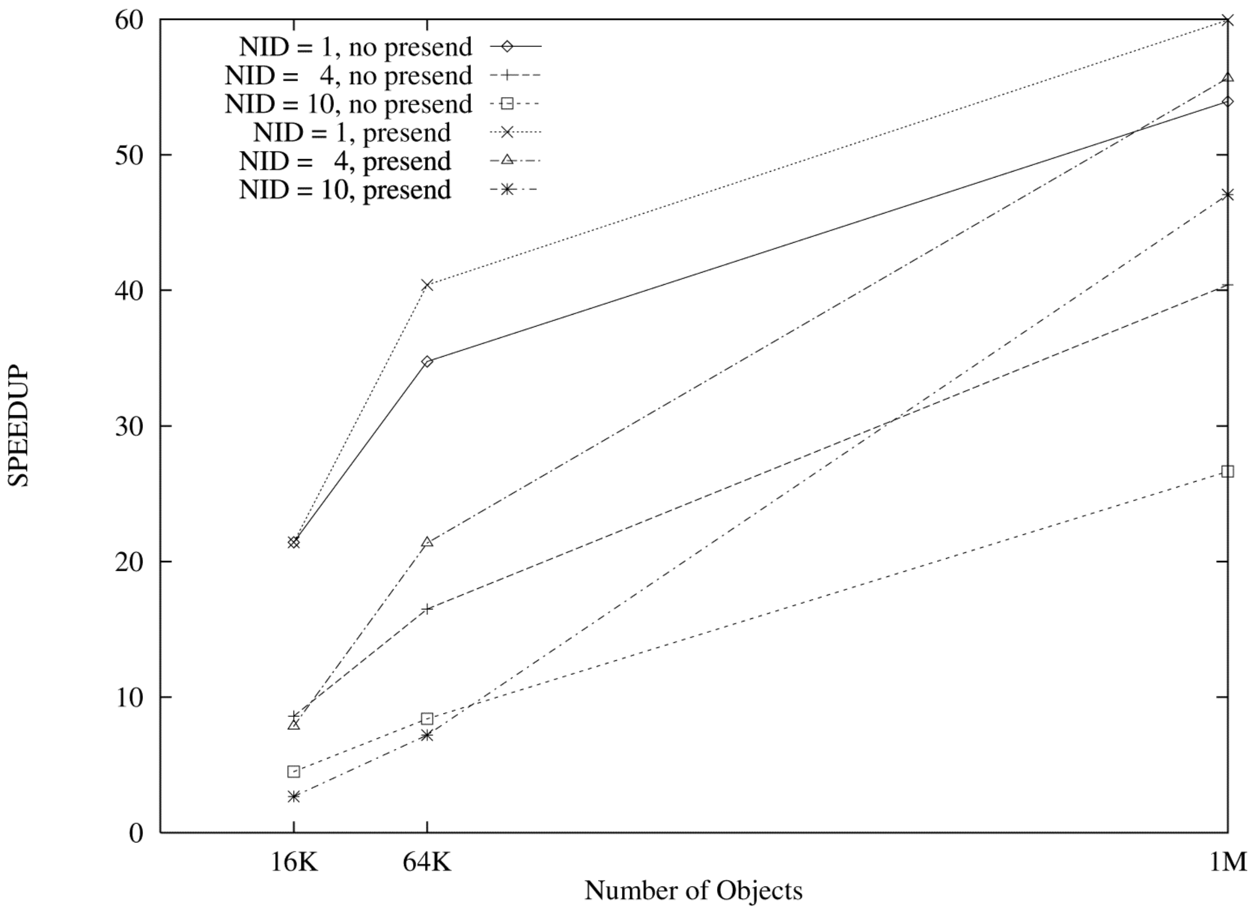

Figure 2.4: Speedup as a function of problem size, for varying Network Interface Delays and pre-send policies.

These behaviors are understood if we consider the ratio of interior objects to boundary objects on each machine. For $128^2 = 2^{14}$ objects that ratio is 7.25; for $512^2 = 2^{18}$ objects the ratio is 31.25; for $1024^2 = 2^{20}$ objects the ratio is 63.25. With 512^2 objects, under pre-sending the offered load to the network interface (1 message every 31.25 event executions, on average) is less than the bandwidth at the network interface (1 message every 5 event executions). The network can handle existing traffic before new traffic is generated, so at the end of the synchronization window we have a shorter delay waiting for communication transfers to complete. Now, in the case of a serial network, the aggregate rate of communication offered to the network is 16 messages every 31.25 event executions on average, just over 1 in 2. The network is capable of handling only 1 in 5, and so a backlog is created; pre-sending only serves to overload the network. That the case of serial communication should perform so "well" can also be explained. On each machine, for every 31.25 events executed, 5 time units of communication work are generated. But, as this is serialized among 16 machines, we have $5 \times 16 = 80$ units of communication for every unit of computation. The ratio of computation to communication is then 0.39, which, if viewed as a processor's "efficiency", predicts a speedup of 25, which is what we observe.

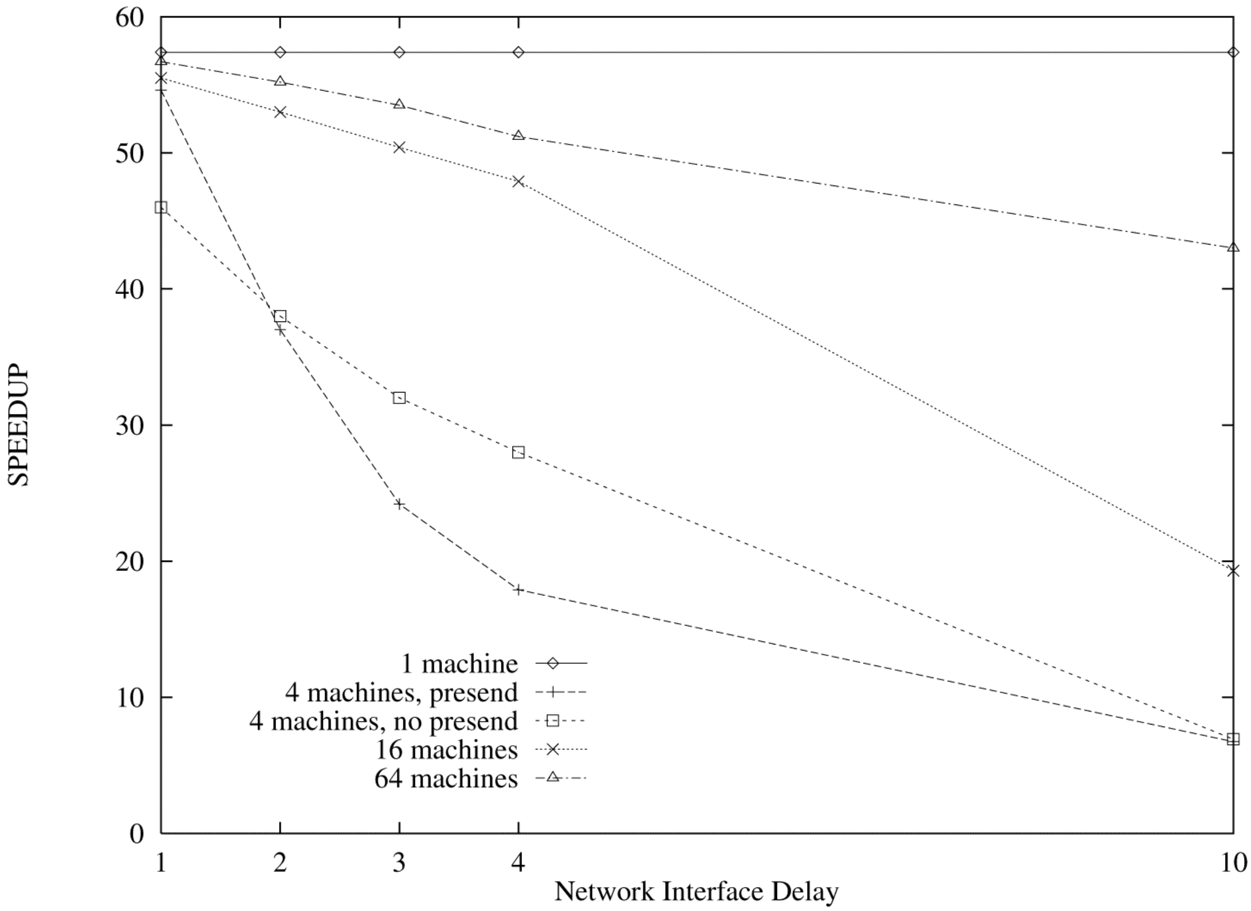

Figure 2.5: Comparison of different architectures as the Network Interface Delay varies.

Evidently there is interplay between NID and computation to communication ratio we should explore; under our partitioning assumptions we control this ratio through the number of objects simulated. Figure 2.4 describes the results of such a study. The graph plots number of objects versus speedup, with each line tracking changes due to increasing problem size for a fixed NID and pre-send parameter. This graph suggests that there is a "cross-over" point where pre-sending begins to be advantageous. We see that good performance is possible, but that relatively poor performance is also possible. It should come as no surprise that the difference boils down to the computation to communication ratio. Higher NID values can be overcome only by a larger number of objects. However, IDES supports simulations with millions of objects, on an architecture similar to that modeled here. These simulations have "locality" of communication, similar to that modeled here.

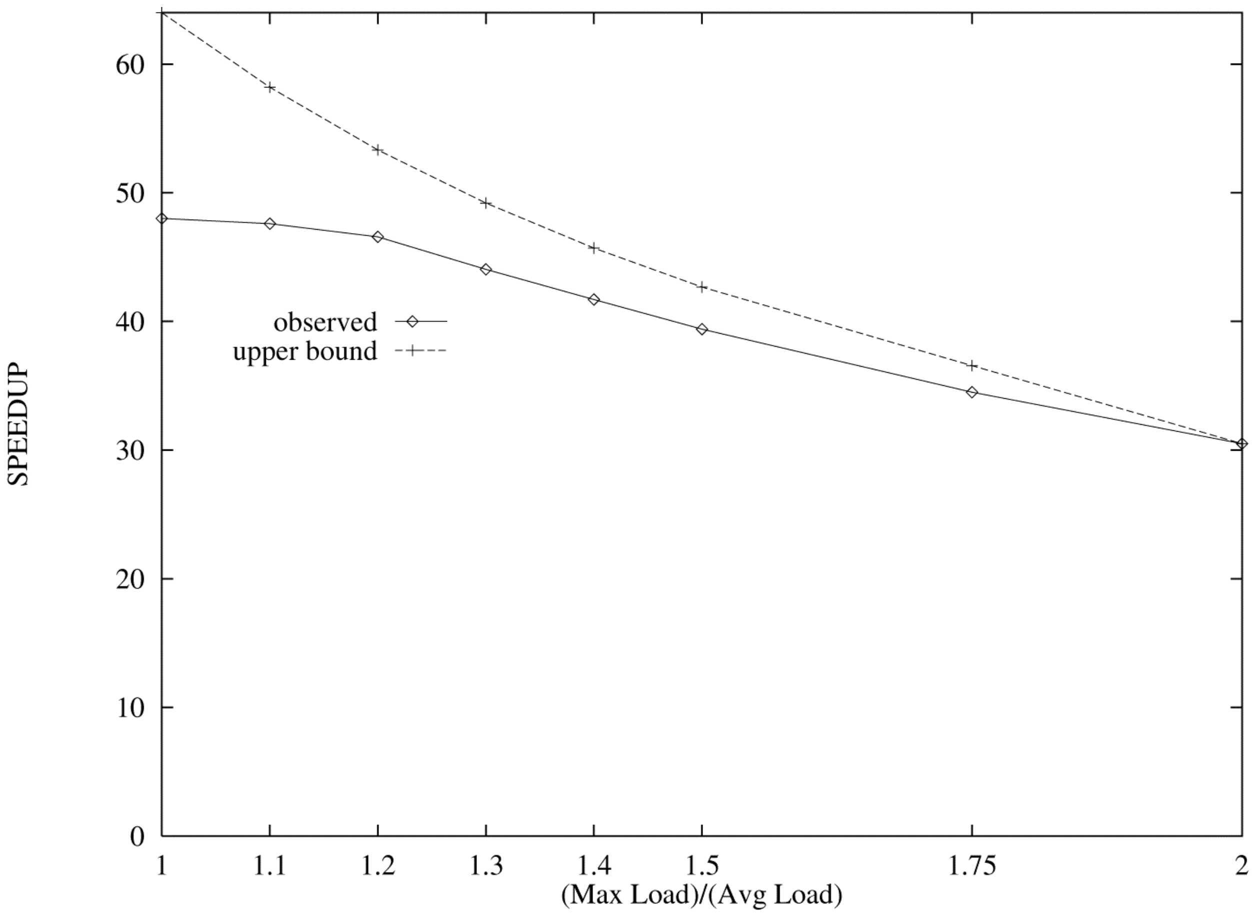

Figure 2.6: Performance as a function of load imbalance.

We next turn to an experiment that examines how architectural characteristics may affect performance. For this study we considered 512^2 objects, balanced workload, present messages, and NID costs of 1, 2, 3, and 4. We considered four architectures that allocated 64 processors to machines in different ways. A 1 machine architecture is just a large shared-memory machine; a 4 machine architecture is a small cluster of medium-scale multiprocessors; a 16 machine architecture is a medium sized cluster of small-scale multiprocessors; a 64 machine architecture is a large cluster of ordinary processors. Figure 2.5 depicts the results. The most striking feature of this graph is how performance of the 16 processors/machine system degrades with increasing NID, and also how the performance of the 16-machine model plummets between NID = 4 and NID = 10 (the performance for not presending is actually worse in this case). For both the 4-machine and 16-machine architectures the single network interface per machine is a bottleneck. This data reminds us of how critical it is to assess a problem's offered network load with respect to the ability of the network to carry that load.

It is also interesting to note that for smaller NID the single processor per machine performance is slightly better than the 4 processors per machine performance. The increased locality of reference achieved by the 16-machine system is overcome by the serialization of communication at the network interface. Finally, somewhat surprisingly, the performance of the "ideal" shared memory machine is not markedly better than that of the 64-machine or even 16-machine architectures.

Next we consider the sensitivity of performance to variations in workload balance. For this experiment we used a base-line system of 512^2 objects, parallel communication network, 64 processors on 16 machines, and a NID value of 4. We then varied the ratio of most heavily loaded processor to average processor load between 1 and 2.0. Figure 2.6 illustrates the results. This graph plots both the predicted speedups, as well as the best possible speedup one can obtain given that level of load imbalance (64/1 ratio). It is interesting to note there is relatively little sensitivity as the load balances moves away from perfect balance. Only in the region of 1.3 or so does the curve begin to behave as one might expect. The flatness of the curve is emphasized by comparison with the best possible performance given the load imbalance parameter.

To understand this behavior we looked more closely at the data. In the case of perfect load balance, in an average window an average processor executed approximately 15% more events than it ultimately committed. The "extra" events amount to 5.5% of the window's duration. Then, it spends 11% of the window's duration completing the min-reduction to establish the window size, and then another 3% of the window's duration waiting for the data messages to be completely delivered. Thus, for 19.5% of the window the average processor is engaged in activities that a serial simulator would not. For the most part, these percentages explain the speedup of just under 50. The main contribution to this overhead is the min-reduction wait, and this wait is due to load imbalance that is inescapable given that the workload in this model is stochastically driven. While the number of objects assigned to each processor is the same, the number of events committed by each processor is not. The 15% extra events executed are not so much a cost as a measurement of the time it takes in BTB to discover the end of the synchronization window. A similar analysis on the data for a load imbalance factor of 1.75 shows that in an average window an average processor spends 14% of the window executing events that are not committed, 30% of the window in the minreduction logic, and 3% of the window waiting for data messages to be delivered. These figures sharply reveal the effects of load imbalance.

It is interesting to note that at a load imbalance factor of 2, the performance is close to the theoretical optimum. By this point performance is dominated by the differences in mean event generation rates; in the balanced case load imbalance was caused by stochastic variance. There is a lesson in this data—that in a stochastic simulation one may be able to tolerate a significant degree of load imbalance due to some imbalance in activity rates. Achieving perfect balance in event generation rates may yield little performance gain if the workload is within 10 or 20 percent of being balanced.

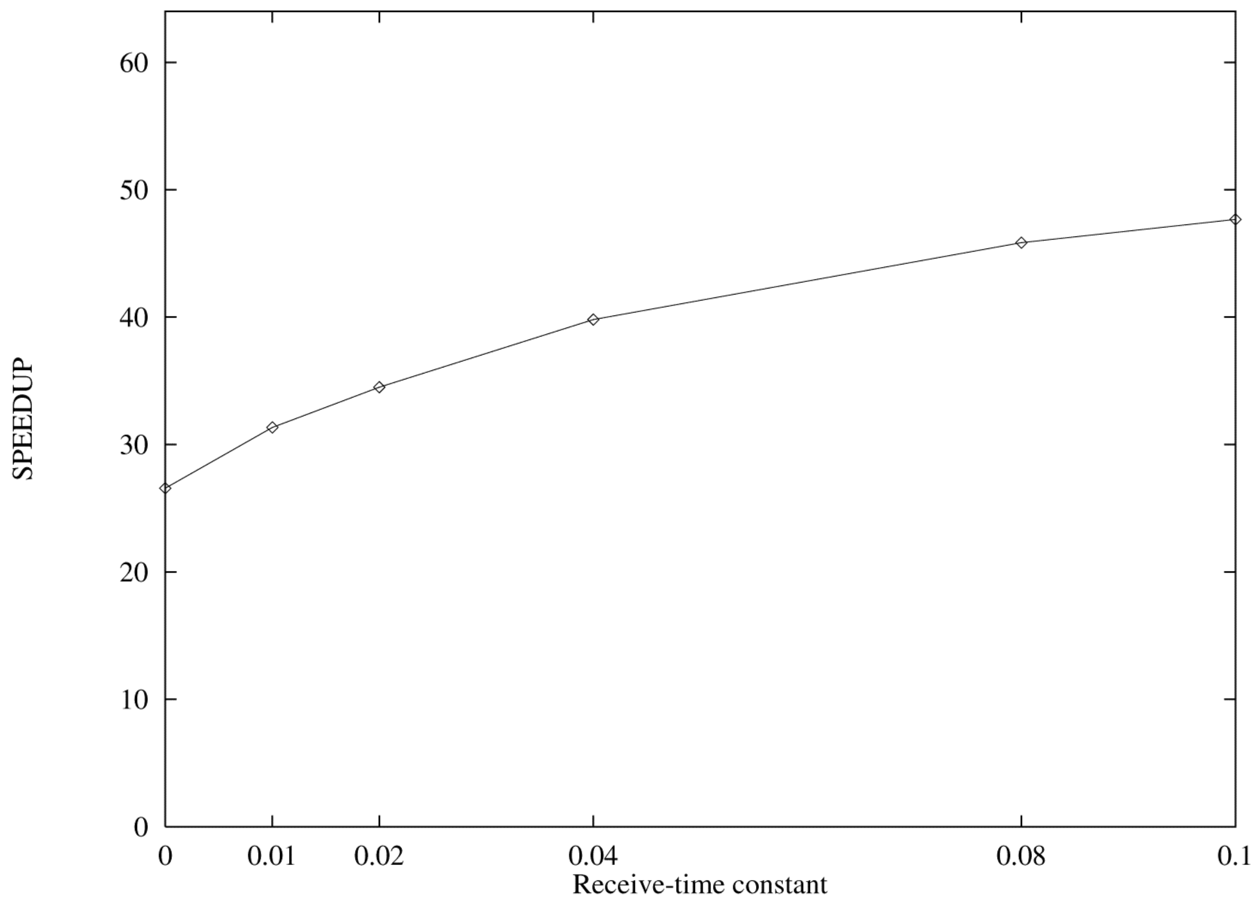

Figure 2.7: Sensitivity to the constant portion c of the send/receive-time gap random variable t.

A last experiment (Figure 2.7) looked at sensitivity to the constant portion c of the send/receive-time gap random variable t. In applications of YAWNS it has been observed that a small constant can improve performance a great deal over that of $c = 0$. We observe the same is true here. We again consider the base system of 512^2 objects, 16 machines, 64 processors, an NID of 4, and pre-sent messages. Varying c from 0 to 0.1 we predict a performance gain (over $c = 0$) of 30% using a constant, 0.02, that represents less than 5% of the mean of the distribution; an 80% gain is obtained with $c = 0.1$ (less than 10% of the mean). Some constant might be deemed essential, but it can be quite small to achieve significant performance gains.

3 THE IDES SYSTEM

We believe that the effort we applied in developing analytic and simulation models of IDES helped us to understand much more deeply how such a system must operate, and the sort of performance sensitivities we could expect from the system once built. Armed with this confidence, we proceeded to implementation. This section describes considerations in the design and implementation of the IDES parallel simulation system. IDES is a Java-based parallel/distributed simulation system designed to support the study of complex large-scale enterprise systems. Using the IDES system as an example, we discuss how anticipated model and system constraints molded our design decisions with respect to modeling, synchronization, and communication strategies.

3.1 SYSTEM DESIGN GOALS

To motivate the IDES system design, consider an example domain: simulation of a U.S. Health Maintenance Organization (HMO). The IDES design was governed by three goals. The first goal deals with the structure of the simulation framework to express the systems to be modeled: link low-level, complex submodels with high-level, policy driven resource allocation techniques to perform cost / benefit trade-off analyses. In the HMO example, each patient is modeled with complex disease processes represented by differential equations—including risk for coronary artery disease. Medical treatment policies interact with disease models to affect the health outcome of patients.

The second goal mandates a type of question the simulation model must be able to answer. Using IDES, we want to study the use of screening techniques to detect an otherwise invisible system deterioration, itself a contributor to a catastrophic failure we would like to prevent. In the HMO example, we would say the early detection and treatment of coronary artery occlusion may extend life and saves later costs when heart failure might otherwise occur.

The third and final goal specifies the portability of the system: development of simulation models using IDES should be within the reach of systems analysts, and support deployment across heterogeneous computing architectures. IDES runs on single-processor systems, networks of workstations, and multiprocessor computers with shared or distributed memory. In addition, IDES incorporates a web-based interface for distributing simulation subcomponents across the enterprise network.

In support of these goals we have developed the IDES framework. IDES is an object-oriented simulation system capable of supporting complex, massive model, parallel discrete event simulations transparently across heterogeneous platforms.

3.2 SYSTEM CONSTRAINTS

In support of these design goals, a number of system constraints had to be overcome. First and foremost, IDES had to be capable of hosting massive models with relatively large state. The example HMO model includes ten million patients and one hundred regional hospitals and facilities. Enterprise simulations evolve differently than more traditional parallel simulation models such as queuing and PCS networks. For example, simulation entity behavior is not governed by a simple draw on a random number stream, but through the evaluation of complex, coupled state-evolution equations. Hence, the difficulty of extracting lookahead discourages the use of a purely conservative protocol.

Since the data state of each component is large, we use multiple machines to acquire the memory needed. While a conservative approach to synchronization could use less memory than an optimistic approach, lack of lookahead limits the effectiveness of conservative synchronization. Consequently very large state coupled with lack of lookahead motivates use of Breathing Time Buckets (BTB) developed by Steinman (1992) to constrain optimism. Furthermore, sheer model size and portability concerns motivated investigation of impact of architecture on performance.

The state of simulation entities is computationally complex. In the HMO example, evaluation of complex disease models is computationally expensive. Parallelism is evident with a large population.

3.3 SYNCHRONIZATION

Synchronization is generally viewed as the key source of difficulty when executing discrete-event simulations. Conservative synchronization methods ensure that every bit of computation executed contributes directly to the final simulation state; optimistic methods support speculative computing where some computations may ultimately be undone. The task of building a parallel simulation framework is understandably easier with a conservative approach. However, there is ample evidence that reasonable performance can be achieved under conservative synchronization only if there is easily extracted lookahead in the simulation model. This simply means that without a great deal of effort it is possible to examine the state of a submodel (the term we'll use to identify that portion of the simulation model that is cohesive in the sense that all simulation work associated with a submodel will be done by the same CPU) and find a lower bound on the time when next that submodel performs some action that affects the state of another submodel. Dissemination of lookahead provides the slack needed between processors that permits them to make forward progress without concern for so-called straggler messages (messages with time-stamps less than the recipients local simulation clock).

Our initial intent was to use a synchronization protocol based on YAWNS by Nicol (1989, 1993). YAWNS is conservative, and when suitable lookahead is available, is provably scaleable. However, as we studied the class of model problems we began to see that lookahead would not be easy to extract without requiring the IDES user to provide more information about the model than we thought the user would typically care to provide. Consider again the HMO model. A patient's risk profile with regards to, say, heart disease, is dependent upon a number of risk factors including life-style choices, family history, and known health problems within ones family. A differential equation describes the probability distribution of the time of next heart attack, as a function of those risk factors. If any of those risk factors were to change, a heavy-weight computation would be required to recompute the probability distribution. The sort of lower-bound calculation needed to compute lookahead would have to identify the worst-case combination of risk factor values and assume they simultaneously changed to this worst case scenario, and then compute a worst-case time-to-heart-attack distribution. The only alternative is to require the modeler to provide this sort of worst case information (at the risk of the modeler being wrong!). We eschewed those constraints in favor of a limited form of optimism that constrains the sort of large-scale memory consumption that general Time Warp simulation is capable of requiring.

We next considered the Breathing Time Buckets (BTB) synchronization approach, as it is essentially an optimistic version of YAWNS. While being optimistic, it ensures that messages between submodels are "correct" in the sense that they will not be canceled. In its simplest form, BTB works as follows. Simulation objects synchronize at points in simulation time (the determination of which is the point of the protocol). At a synchronization point, messages are exchanged between submodels; as these messages are correct, they can be incorporated into their recipients' event lists. Next a submodel executes events on its event list in time-stamp order, performing state-saving. As messages to other submodels are generated, these are buffered but their so-called *receive-times* are noted, the times when the message affects the recipient (as opposed to the time when the sender sends it, which may be different). A submodel tracks the minimum receive-time of any message it generated but has not yet delivered. At the point when the time of next event is greater than or equal to the minimum such receive-time, the submodel has reached its *local event horizon*. BTB defines the next synchronization point as the minimum local event horizon among all submodels, this is called the *global event horizon*. The global event horizon essentially establishes the least next time when an as-yet-unknown message can arrive at a submodel and change its state. Therefore, all computation up to the global event horizon is known to be "good" in that even though computed speculatively, it did not depend upon a message from another submodel. Of course, a submodel may have been advanced beyond the global event horizon, and so (at least conceptually) is rolled back to the global event horizon.

A naive way of determining the global event horizon is to have each submodel execute all the way until reaching its local event horizon, and then engage in a global minimum-reduction operation to identify the least such. This would actually maximize the amount of memory used for state-saving in a BTB approach, in that each submodel would be executed as far as could be possible, saving state the entire way. Clearly, to reduce state-saving costs one needs to disseminate local event horizons as they are discovered. Towards this end we developed an algorithm—the preemptive min-reduction—to attempt to identify and distribute the global event horizon quickly.

In a normal reduction a processor offers a value to the reduction operator and then blocks until all processors have offered values and the reduction is performed. A processor interacts with a preemptive min-reduction somewhat differently. Each processor has a "working minimum" in the case of BTB the least observed receive time on generated messages. As the computation progresses, the working minimum changes in a monotonically non-decreasing fashion.

The reduction framework in a processor maintains a "partially reduced'' value, initially infinity, to reflect the minimum value reported to that processor in the course of the preemptive-reduction. Periodically (say, after each event) a processor compares its time of next event with the partially reduced value. If the former value is smaller, the processor's progress has been preempted by knowledge of the existence of a local event horizon, somewhere, that is smaller than the processor's own. It then engages in the reduction logic, offering the partially reduced value as its own.

It blocks until the reduction is completed and the global event horizon is identified. Alternatively, if a processor reaches its local event horizon without being preempted, it simply engages in the min-reduction. All that is needed to implement this algorithm is user code access to the partially reduced value that is typical in tree-based reduction algorithms. We have based our implementation on the non-committal barrier synchronization by Nicol (1995).

3.4 IDES IMPLEMENTATION

The IDES design has been implemented separately in both C++ and Java. This paper deals exclusively with the Java implementation.

3.4.1 CLASS STRUCTURE

The two main simulation classes are *Entity* and *Message*. All simulation objects are represented by the Entity class which encodes individual state and behavior. Entities communicate with one another by sending Messages which contain routing information as well as message content.

Two additional base classes complete the IDES framework: *EventQueue* and *Router*. In IDES, a simulation is decomposed into a number of submodels, each consisting of a subset of all simulation Entities (Figure 3.1). Each submodel contains an EventQueue and a Router.

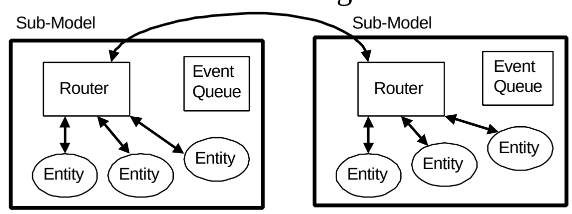

Figure 3.1: IDES model decomposition.

```
protected double wakeup
(double time) {

   // CHECKPOINT THE STATE OF THE OBJECT, AND
   //   UPDATE ENTITY TO THE CURRENT TIME.
   checkpoint(time);
   update(time);

   // PERFORM INTERNAL ENTITY EVENTS.
   performInternalEvent();

   // RESPOND TO EXTERNAL MESSAGES.
   while(!messages_.isEmpty())
     performMessage(messages_.dequeue());

   // DETERMINE TIME OF NEXT WAKEUP.
   return forecast();
}
```

Figure 3.2: *Entity* event processing routine.

Execution of simulation events for Entities on the submodel is controlled by the submodel's EventQueue. The role of the EventQueue is simply to hand the thread of execution control to the appropriate Entity at the appropriate simulation time, by invoking the Entity's *wakeup* routine (Figure 3.2). In this routine, the Entity executes the events that should occur at that time, including response to and sending of Messages if required. It then gives execution control back to the EventQueue, having forecast (Figure 3.3) the time of next wakeup. Hence each entry in the EventQueue consists of an Entity reference and the simulation time at which the Entity should be "woken up."

```
protected double forecast() {

   // CALCULATE EARLIEST INTERNAL EVENT.
   wakeupTime_ = forecastInternal();

   // CALC. EARLIEST MESSAGE RECEIVE TIME.
   if (!(messages_.isEmpty())) {
      double messageTime = messages_.headKey();
      if (messageTime < wakeupTime_)
         wakeupTime_ = messageTime;
   }

   // RETURN EARLIEST TIME.  THE ENTITY WILL
   //   BE WAKEN UP AT THIS TIME.
   return wakeupTime_;
}
```

Figure 3.3: *Entity* forecast.

The Router is responsible for routing and filtering all Messages that are sent to and from the Entities on the Router's submodel. The Router is also responsible for establishing synchronization windows with other Routers in the simulation, according to the algorithm discussed above.

3.4.2 Decomposition Mechanism

Entities are arranged in a hierarchy in which parent Entities are responsible for instantiating child Entities. We refer to the highest level parents as the *top-level Entities*.

For a particular simulation run, each top-level Entity must be assigned to a specific submodel. We implement this mapping as a matrix of size (number of top-level Entities) x (maximum number of submodels allowed). For any top-level Entity, given the number of submodels in the simulation, the corresponding matrix entry identifies the assigned submodel.

Invocation of the IDES executable code instantiates a single submodel to which two arguments must be passed: (1) the total number of submodels in the simulation and (2) the unique identifier for this particular submodel. Each submodel will then instantiate only the top-level Entities that have been assigned to it, based on the matrix described above.

It should be noted that the Entity to submodel assignment is an initial (simulation start-up) assignment only. We do not restrict Entities from migrating from one submodel to another during a simulation run.

3.4.3 Code Distribution

The IDES distribution mechanism is also implemented in Java. At start up, the IDES Server is running on every machine that may be used as a host for the simulation run. The Server's user interface (Figure 3.4) allows the owner of the machine to control the use of the machine by remote IDES Clients. The IDES Client (Figure 3.5) is run by the simulation owner (the "user"). For a simulation run, the user indicates (1) the directory in which the simulation executable code resides, and (2) the machines on which the simulation is to be run. As each machine is selected, the IDES Client contacts it to ensure that the IDES Server is running there, ready to accept transmission of the simulation code.

Upon user command, the IDES Client transmits to each participating Server the following data: (1) the simulation executable code, (2) the identification number for the submodel to be instantiated, and (3) the total number of submodels in the simulation. The Server then invokes the executable on its machine, creating the proper submodel. The Client also sends to each Server the addresses and submodel identification numbers for all other participating machines. This information is passed to the executing submodel whose Router then uses it to establish a communication link to the Router in each of the other submodels. The simulation is now ready to run.

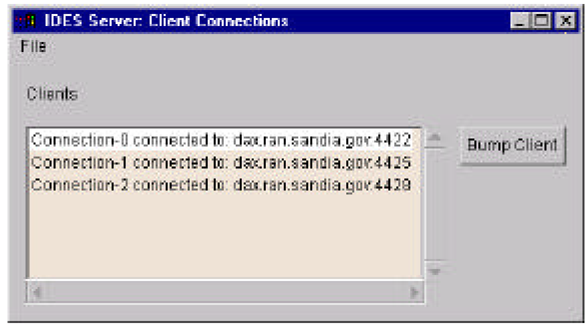

Figure 3.4: IDES code distribution server.

3.4.4 State Saving Mechanism

Within BTB, individual simulation submodels are allowed to optimistically surge forward, speculatively executing events on their events lists in time-stamp order. Since receipt of a message with receive time less than the current event execution time necessitates a state rollback, submodels must perform state saving.

Driven by the need to support massive models and thus limit the amount of saved state, we first considered the naive approach of state saving only once at the window boundary. The simulation would then be allowed to process forward speculatively until detection of the event horizon. With the event horizon determined, all simulation submodels would be rolled back to the beginning of the window and run forward again to stop at the event horizon. While this scheme minimizes the amount of saved state, it necessitates execution of the simulation twice.

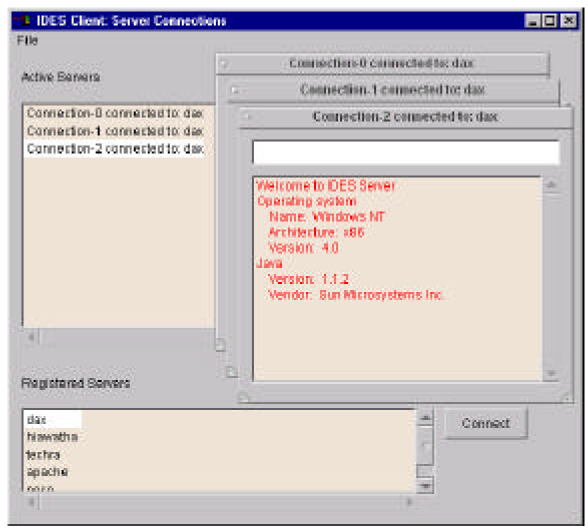

Figure 3.5: IDES code distribution client.

Next we considered going to an incremental mechanism whereby individual state variables are saved as they are changed. However, implementing this scheme in Java appeared complicated and overly taxing on the user of the system. In addition, experiments showed that due to the coupling of state variables in the objects of interest to IDES, execution of a typical event touched most state variables anyway.

```
abstract public class Entity extends Persistent implements
Serializable
{ …}
```

Figure 3.6: *Entity* class declaration.

In the face of these considerations, we implemented what is commonly known as “copy’’ state-saving—see Franks, Gomes, Unger, and Cleary (1997) for a discussion of various state-saving policies. Immediately prior to receipt of a message or processing of an event, the system checkpoints the mutable state of the affected entity. The state saving mechanism relies on the Java implementation of object serialization. All IDES object classes are required to derive from Entity (Figure 3.6), which itself derives from Persistent.

The class Persistent contains the routines for checkpointing and rollback of individual Entity state. This is accomplished through an internal ordering of ByteArrayOutputStreams serialized through an ObjectOutputStream. In the IDES object class hierarchy, all classes from the Entity down are required to implement Serializable (Figure 3.7). The one drawback to this mechanism is the requirement that object images must be restored to a new address. In most cases, the user wants to *update* the state of an existing object with only those variables that could possibly change since the last checkpoint, and not replace all Entity state values completely. In order to accomplish this, our implementation relies on the Serializable mechanism to restore the state of transient (or non-persistent) variables into a new address space. Then a Persistent routine, *clone*, copies the contents of the newly restored object image into the original image.

```
public class
Car extends Entity implements Serializable {
  public Car (Router router,
            String name,
            int dealerId,
            double maintenanceInterval,
            double messageDelay) { }
…}
```

Figure 3.7: *Car* class declaration.

3.4.5 Example Simulation Problem

Our example problem domain is an automobile franchise comprised of *Dealers*, *Owners*, and their *Cars*.

Dealers sell and service Cars. They also on occasion will issue recalls on certain defective Cars they have sold. Services on Cars include both routine maintenance work and recall repairs.

Owners purchase Cars from Dealers. They may request service from any Dealer, but recalls will always be received from the original (selling) Dealer.

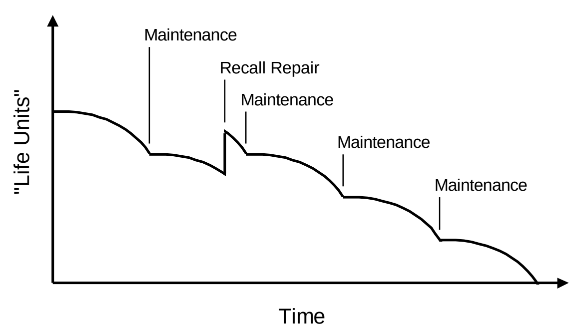

Figure 3.8: *Car* deterioration model.

Cars deteriorate with time (Figure 3.8). Routine maintenance slows the rate of deterioration, but cannot prevent it completely. Defects in Cars can be corrected by recall repair work. The useful life of a Car is affected by the presence of defects and the service work received over the life of the Car. When a Car dies, its Owner purchases a new Car from the same Dealer from which the first Car was purchased.

The following code sample (Figure 3.9) is from the Dealer class, in which a Dealer performs a recall event.

```
private void performInternalEvent () {
  if (time_ == recallTime_) {

    // SEND MESSAGE TO CAR TO BE RECALLED.
    sendMessage(new Message( myId_,
              recallCarId_,
              currentTime_,
              (currentTime_ + 0.5),
              Message.RECALL));
  }
}
```

Figure 3.9: *Dealer* sending a message.

The *sendMessage* routine is used to send a Message to the Car to be recalled. In creating the Message, the sending and receiving Entity identifications, the send and receive times, and the type of the Message must be specified.

```
private void performMessage (Message msg) {
  if (msg.type() == Message.RECALL) {

    // PERFORM RECALL.
    lifeUnits_ += .1;
    if (lifeUnits_ > 1.0)
      lifeUnits_ = 1.0;
  }
}
```

Figure 3.10: *Car* recall message handler.

Response to a received Message is done in *performMessage* (Figure 3.10). The example above is for a Car that has received a recall Message.

After having decoded the recall message, the Car performs the recall—here simply an adjustment of the Car's life units—and then returns immediately to the event-processing loop. Next the Entity must determine the future wakeup time based on pending internal events and messages—a function performed by forecast. Since the recall affected the life units of the Car, and hence the internal state of the Entity, the forecast routine must determine when the next internal Entity event will occur.

```
protected double forecastInternal() {

  // EVALUATE DIFFERENTIAL EQUATIONS
  //   TO DETERMINE PREDICTED DEATH TIME.
  double nextTime = calcDeathTime();

  // SCHEDULE MAINTENACE IF PRIOR TO DEATH.
  if (nextTime > maintenanceTime_)
    nextTime = maintenanceTime_;
  return nextTime;
}
```

Figure 3.11: *Car* forecast internal event.

Forecast internal event (Figure 3.11) calculates the time of next internal event for an Entity. In the simple example given for a Car, the only possible two internal events are the demise of the Car, or a request for maintenance. Once the minimum has been determined, the forecast routine (Figure 3.3) then decides if the next internal event, or receipt of a pending message, will result in the next Entity wakeup.

4 BATCH SIMULATION SYSTEM

In addition to developing a system to support enterprise simulation, we sought to implement a batch simulation system. Our goal utilizing standardized off-the-shelf distributed object and clustering tools, was to develop a batch simulation systems with the capability to run multiple instances of an application distributively across a heterogeneous network of computers. The initial proof of concept demonstration was to support multiple, distributed instances of SPICE running in parallel across a network of homogeneous Linux workstations.

SPICE is an electrical circuit simulation package developed by UCB. Unsupported versions of SPICE are available under public licenses across the network. Many commercial vendors have wrapped and enhanced the SPICE application into products with graphical user interfaces. The majority of the electrical circuit simulation performed at Sandia is currently done using a commercial product, MicroSim PSpice (based on SPICE v2G6 an older FORTRAN/C version), which runs single-threaded under Microsoft Windows NT.

A SPICE circuit simulation takes as input a net list, and generates one or more output files. There are two particularly demanding types of SPICE circuit simulations performed at Sandia: (1) a number (hundreds) of iterations of SPICE (each a 5-10 minute problem) varying individual parameter values in a single input net list; and (2) single execution of SPICE (a single two day problem) using one complex input net list. The second application could only be sped-up through use of a multi-threaded version of SPICE on a multi-processor machine—and this was not the focus of our development. The focus of this effort is the development of software to demonstrate a speed-up of the first circuit simulation application area in a homogenous environment, and later to extend this to an infrastructure supporting general batch oriented simulation in heterogeneous environments.

Our approach was to utilize standardized, off-the-shelf distributed object and clustering tools, to develop the capability to run multiple instances of an application distributively across a heterogeneous network of computers. Seated at a single computer terminal, a user should supply: (1) an application executable for each computer platform supported on the heterogeneous network, (2) one or more input files, and (3) a location for collection of the application results. A general solution would include the following steps: (1) executable distribution to remote computing platforms; (2) configuration and input file management and distribution; (3) remote model execution; and (4) program output capture and consolidation. The distribution of input files, remote execution of multiple instances of an application, and the subsequent consolidation of output should be details beyond the user's concern.

At the outset of this work we evaluated publicly available batch systems and found none that would meet our needs and so embarked on development of a batch system from scratch. Despite the many man years of effort needed to develop other batch systems, we believed that the project could be completed in the allocated time by using state-of-the-art programming techniques (in particular, CORBA to provide a distributed object-oriented programming environment) and by limiting our attention to the needs of the particular project. Indeed, rapid progress on the CORBA based batch system was achieved and will be discussed in more detail below. However, several factors led us to reconsider the use of another batch system with publicly available source code. The first factor was the immaturity of many of the CORBA implementations. Second, one of the publicly available batch systems, DQS, seemed much more promising than it did on the first evaluation and we discovered another batch system, PBS, that was then undergoing beta testing and seemed to meet all of our requirements. The final factor was the desire by the Sandia SPICE team to begin immediate use of IDES. The CORBA based system could not yet schedule parallel jobs and thus could not yet be deployed. Thus, our implementation strategy was revised to immediately concentrate on modifying DQS to schedule jobs. Our experiences with DQS will be summarized below. We plan to follow the development of PBS and CORBA to determine if it would be desirable to switch at some later date to PBS or resume development of our own CORBA based system.

4.1 CORBA BATCH SYSTEM

The Common Object Request Broker Architecture (CORBA) permits objects residing on one node in a network to be accessed by processes on different nodes in a completely transparent fashion. The process that implements the objects is the server for that object and processes that remotely invoke the methods of that object are clients of that server. CORBA provides a clean, simple way to implement client/server systems and is an ideal way to implement a batch scheduling system.

In a batch scheduling system there are several nodes that provide computing resources. In our CORBA batch system, each of these nodes has a server, the Machine Server, for objects that provide information about the machine and start and manage jobs for that machine. A central batch scheduling process acts as the server, the Batch Server, for objects that maintain the queue of jobs as well as the job objects themselves. In addition, utility programs which act as clients of these objects are used by users to submit and monitor their job. Furthermore, the Machine Server acts as a client of the Batch Server when it announces itself to the system and when in obtains information about jobs to run. The Batch Server acts as a client of the Machine Server when it obtains information about the resources available on that machine. In the CORBA approach, the specifications of related object interfaces are grouped together in a module.

The objects provided by the Machine Server have all of their interfaces specified in the Machine module which consists of the Info, Spawner, and Machine classes. Objects of the Info type provide information about a machine’s resources, such as the number of processors and the amount of memory. Objects of type Spawner create and monitor batch jobs. Each Machine Server has one object of type Machine, which is responsible for registering the machine with the Batch Server and creating Spawner objects.

The interfaces for objects provided by the Batch Server are specified in the Batch module, which consists of the Job, SimpleJob, Resource, and Batch classes. The Job class is an abstract base class for batch jobs. SimpleJob is a specialization of this class for batch jobs that simply run a command specified by the user. Objects of class Resource are used to keep track of the resources used by a particular user of the batch system. The Batch Server has one object of type Batch that maintains the queue of jobs, schedules jobs, and provides information about the batch system to users.

The initial implementation of the machine and batch servers were in C++. However, the CORBA/C++ software available on Linux, at that time, was unstable. Thus we rewrote the servers in the interpreted language Python. All along the graphic user interface, which acted as a client to both the Batch Server and Machine Server, was written in Python.

As the development continued difficulties with the CORBA software available for Linux were revealed. All of the CORBA implementations for Linux were single-threaded and supported no means to provide mutually exclusive access to certain regions of code. Unfortunately, in a distributed system such as this, where multiple users may be simultaneous clients of the Batch Server, there are really multiple threads of control.

The problems arise when a server acts as a client of another job, for example, when the Batch Server creates an object to spawn a job on another machine using its machine server. While the Batch Server is waiting for a reply from the Machine Server it must continue processing requests. For example, the machine server will need to obtain the job and its characteristics from the Batch Server while it is creating the process. However, suppose a user requests that the job be canceled while the job is being started. The batch object's data structure can be placed in an inconsistent state resulting in an eventual failure. The preferred solution is to use mutual exclusion locks to protect sensitive areas of code. Since these locks were not available to us, we pursued the alternative of carefully coding the application so that methods of remote objects will not be invoked during critical sections of code. Unfortunately, this defeats somewhat the purpose of an object-oriented approach, since knowledge of the details of each object's implementation is needed to write correct code to use it.

4.2 Distributed Queuing System

The Distributed Queuing System (DQS) is a publicly available batch queuing system related to the widely used Network Queuing System (NQS). It is a complete rewrite of NQS and provides the additional capability of being able to run parallel jobs across some or all of the nodes in a cluster of machines.

Although some bugs have arisen in DQS on our system, after applying our fixes DQS seems exceptionally stable. Furthermore, the DQS development team has been quite responsive in applying our fixes to their code so that the bugs will not affect us in future releases.

The main problem with DQS is its simplistic scheduling system for parallel jobs. When a parallel job is at the top of the queue it can only run if the minimum number of processors it requests are available. Otherwise, the next job will be considered and so on. If one of these jobs only requires as many processors as are available, then it will be run. Thus, the parallel job may be prevented from running indefinitely. We plan to monitor the performance of DQS to determine if and how the scheduling algorithm needs to be rewritten.

5 CONCLUSIONS

The IDES project at the Sandia National Laboratories developed a simulation environment for large-scale, fine-grained problems. IDES goals included portability over various architecture types. Focusing on the Breathing Time Buckets synchronization protocol, we have developed a simple model of performance; the purely analytic model is expressed in terms of state transition equations that can be solved efficiently, numerically. Performance reported is based on a discrete-event simulation of the model. We have developed a new algorithm—the preemptive min-reduction—for quickly detecting the end of the BTB synchronization window. We have considered an alternative strategy for managing communication—to pre-send all messages and have the receiver filter out the risk-free ones once the window edge is known. We have looked at the sensitivity of performance to key parameters such as problem size and communication delay, and have confirmed that our alternate communication strategy can provide significant performance gains. For the scale of problems and architectures anticipated for IDES, we see that good performance will likely be achieved.

IDES provides an object-oriented foundation for simulation that is applicable to all of Sandia's simulation projects, for example, (1) enterprise modeling—stockpile maintenance, (2) quantum chemistry codes—materials aging, and (3) systems studies—gamma-ray transport. The availability of this software system will increase the scale at which a large class of real-world systems can be modeled. In addition, IDES protects the investment in the construction of such models by providing, (1) a standardized API with which to quickly model complex systems; (2) immediate performance gains through parallel simulation without involving the system's user; and (3) a portable means of developing system simulation software, eliminating the user's dependence on a single hardware platform. IDES addresses Sandia's need for a robust, portable, scaleable simulation system which will span more than a single project, and which can be utilized throughout the company.

REFERENCES

R. Bagrodia, W. Liao, 1994. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. In *IEEE Transactions on Software Engineering*, Vol. 20, No. 4, 225-238, April, 1994.

K. Chandy, R. Sherman, 1989. The Conditional Event Approach to Distributed Simulation. *SCS Multiconference on Distributed Simulation*, The Society for Computer Simulation.

R. Felderman and L. Kleinrock, 1991. Bounds and approximations for self-initiating distributed simulation without lookhead. *ACM Transactions on Modeling and Computer Simulation*, 1(4), October 1991.

A. Ferscha, 1995. Probabilistic adaptive direct optimism control in time warp. In *Proceedings of the 1995 Workshop on Parallel and Distributed Simulation*, pages 120-129, Lake Placid, NY. The Society of Computer Simulation.

S. Franks, F. Gomes, B. Unger, and J. Cleary, 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 72-79. IEEE Computer Society Press.

M. Gupta, A. Kumar, and R. Shorey, 1996. Queueing models and stability of message flows in distributed simulators. In *Proceedings of the 1996 Workshop on Parallel and Distributed Simulation*, pages 162-169, Philadelphia, PA. The Society of Computer Simulation.

W.D. Hillis and Jr. G.L. Steele, 1986. Data parallel algorithms. *Communications of the ACM*, 29(12):1170-1183, December 1986.

D. Nicol, 1991. Performance bounds on parallel selfinitiating discrete-event simulations. *ACM Transactions on Modeling and Computer Simulation*, 1(1):24-50, January 1991.

D. Nicol, 1992. Conservative parallel simulation of priority class queueing networks. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):294-303, May 1992.

D. Nicol, 1993. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304-333, April 1993.

D. Nicol, 1995. Non-committal barrier synchronization. *Parallel Computing* (21): 529-549.

D. Nicol, M. Johnson, A. Yoshimura, and M. Goldsby, 1997. Performance modeling of the IDES framework. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 38-45. IEEE Computer Society Press.

D. Nicol, C. Michael, P. Inouye, 1989. Efficient aggregation of multiple LPs in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, 680-685.

M. Raynal, 1988. *Distributed Algorithms and Protocols*. John Wiley and Sons, New York.

J. Steinman, 1992. SPEEDES: A multiple synchronization environment for parallel discrete-event simulation. In *International Journal in Computer Simulation* (2): 251-286.

J. Steinman, 1994. Discrete-event simulation and the event horizon. *In Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, pages 39-49, Edinburgh, Scotland. The Society of Computer Simulation.

BIBLIOGRAPHY

Bagrodia, Rajive L., Chandy, K. Mani, and Misra, Jayadev, "A Message-Based Approach to Discrete-Event Simulation", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, 654-665, June 1987.

Bershad, Brian N., “The PRESTO Users Manual”, University of Washington, October 1991.

Chandra, Rohit, Gupta, Anoop, and Hennessy, John L., “Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language”, *IEEE Computer*, February 1994.

Edwards, G., and Sankar, R., "Modeling and Simulation of Networks Using CSIM", *Simulation*, Vol. 58, No. 2, 131-136, February 1992.

Egdorf, H.W., and Painter, Steven W., “An Object-Oriented Methodology for Discrete-Event Simulation Tasks: Requirements, Functional Specification, Design, Implementation”, Los Alamos National Laboratory.

Engler, Dawson R., Andrews, Gregory R., and Lowenthal, David K., “Filaments: Efficient Support for Fine-Grain Parallelism”, The University of Arizona, Tucson, Arizona.

Fisher, Joseph A., "Object Oriented Simulation Tools for Discrete-Continuous, Stochastic-Deterministic Simulation Models", Oregon State University, Master of Science Thesis, August 24, 1992.

Fishwick, Paul A., “SimPack: Getting Started with Simulation Programming in C and C++”, University of Florida, Department of Computer & Information Science.

Freeh, Vincent W., Lowenthal, David K., and Andrews, Gregory R., “Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations”, *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Usenix Association, November 1994.

Fujimoto, Richard M., "Parallel Discrete Event Simulation", *Communications of the ACM*, Vol. 33, No. 10, 30-53, October 1990.

Fujimoto, Richard M., Tsai, Jya-Jang, and Gopalakrishnan, Ganesh C., "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp", *IEEE Transactions on Computers*, Vol. 41, No. 1, 68-82, January 1992.

Grunwald, Dirk, "A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System", Technical Report CU-CS-552-91, Dept. of Computer Science, University of Colorado, November 1991.

Heidelberger, Philip, and Nicol, David, “Conservative Parallel Simulation of Continuous Time Markov Chains Using Uniformization”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 41, No. 8, August 1993.

Keppel, David, “Tools and Techniques for Building Fast Portable Threads Packages”, University of Washington, Technical Report UWCSE 93-05-06.

Little, M. C., and McCue, D. L., "Construction and Use of a Simulation Package in C++", technical report, Dept. of Computing Science, University of Newcastle upon Tyne.

Lubachevsky, Boris D., "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communications of the ACM*, Vol. 32, No. 1, 111-123, January 1989.

Misra, Jayadev, "Distributed Discrete-Event Simulation", *Computing Surveys*, Vol. 18, No. 1, 39-65, March 1986.

Nicol, David M., "Problem Characteristics and Parallel Discrete Event Simulation", Book Chapter 1, Dept. of Computer Science, College of William and Mary.

Nicol, David M., Fujimoto, Richard M., "Parallel Simulation Today", College of William & Mary, Department of Computer Science, Georgia Institute of Technology, College of Computing, Annals of Operations Research, Vol. 53, 249-285, 1994.

Nicol, David, and Heidelberger, Philip, "On Extending Parallelism to Serial Simulators", technical report, Dept. of Computer Science, The College of William and Mary, November 28, 1994.

Nicol, David M., and Mao, Weizhen, "Automated Parallelization of Timed Petri-Net Simulations", technical report, Dept. of Computer Science, The College of William and Mary.

Reynolds, Paul F., Jr., "A Spectrum of Options for Parallel Simulation", *Proceedings of the 1988 Winter Simulation Conference*, M. Abrams, P. Haigh and J. Comfort (eds.), 325-332, 1988.

Reynolds, Paul F., Jr., Pancerella, Carmen M., and Srinivasan, Sudhir, "Design and Performance Analysis of Hardware Support for Parallel Simulations", technical report, Dept. of Computer Science, School of Engineering and Applied Science, University of Virginia.

Righter, Rhonda, and Walrand, Jean C., "Distributed Simulation of Discrete Event Systems", *Proceedings of the IEEE*, Vol. 77, No. 1, 99-113, January 1989.

Rogers, Ralph V., "Synchronization of Autonomous Objects in Discrete Event Simulation", Washington, DC : National Aeronautics and Space Administration; Springfield, Va. : National Technical Information Service, distributor, 1991].

Rothenberg, Jeff, "Object-Oriented Simulation: Where Do We Go from Here?", Santa Monica, Ca. : Rand Corp., 1989.

Schmidt, Douglas C., "The Adaptive Communication Environment, Object-Oriented Network Programming Components for Developing Distributed Applications", University of California, Irvine, Department of Information and Computer Science.

Schwetman, Herb, "CSIM17: A Simulation Model-Building Toolkit", Mesquite Software, Inc.

Srinivasan, Sudhir, and Reynolds, Paul F., Jr., "On Critical Path Analysis of Parallel Discrete Event Simulations", Computer Science Report No. TR-93-29, Dept. of Computer Science, School of Engineering and Applied Science, May 25, 1993.

Still, Charles H., "Portable parallel computing via the MPI1 message-passing standard", *Computers in Physics*, 8(5):533-539, Sep./Oct. 1994.

Sunderam, V. S., and Rego, Vernon J., "EcliPSe: A System for High Performance Concurrent Simulation", *Software—Practice and Experience*, Vol. 21, No. 11, 1189-1219, November 1991.

Wonnacott, Paul, and Bruce, David, "The Design of Apostle—A High-Level, Object-Oriented Language for Parallel and Distributed Discrete Event Simulation", Defense Research Agency, Malvern, Worcestershire, United Kingdom, 1995.

Appendix A: PDES System Design Issues

In developing IDES we investigated a number of parallel, discrete-event simulation system level design issues and their relation to anticipated problems IDES would be used to solve. The following sections detail those issues.

A.1 An Object Oriented Model for Parallel Programming

A convenient and much-used model for parallel programming pictures the parallel program as a collection of processes which send messages to each other. It is possible to put any such process into a normal form in which the process's code is broken into sections that are triggered by the receipt of external messages and that execute to completion.

Implementing a program in such a form has certain advantages. Since each section of the code executes to completion (after which the process waits for another external message), there is no need for the process to have its own stack. All the state that must be preserved for the process is contained in the process's explicit local variables. For this reason, memory usage and context-switching overhead can be lower than with a standard implementation of processes or threads in which each has its own stack.

Such an implementation maps directly onto an object-oriented model in which the external messages correspond to method invocations. In order to give the model the generality it needs to express any parallel program, it is necessary to allow the object to disable its own methods selectively. When a method is disabled, messages corresponding to it are queued rather than consumed. Note that Bagrodia's COMPOSE system, which asks the user to write according to the OO model in the first place, sacrifices no generality by doing so. COMPOSE associates each method with a boolean guard that disables the method when it is false. COMPOSE is specialized for PDES, since it delivers messages in timestamp order.

The implementation can be made with a single thread of control on each node of the parallel system (so there is no context-switching per se). Let the thread run a "daemon" that invokes the methods of the objects that implement (or have the same effect as) the processes of the parallel program. The daemon also acts as a message router. A message sent by an object goes first to the daemon on the node on which the object is running. If the message is for another object on that node, the daemon enqueues it and later passes it to the receiving object by invoking one of its methods. If the message is for an object on another node, the daemon forwards the message to that node, where the local daemon enqueues it and eventually passes it to the object.

In order to be able to write a daemon which can run any application and does not have to know all the details about the methods offered by a particular set of application objects, we specify that all objects have a method

```
void receiveMessage (Message&);
```

which is never disabled, where Message is the base class for all the different messages the processes can send. An object's receiveMessage method analyzes the received message and calls a private method of the object in order to process it. It is these private methods which must be capable of being disabled. When such a method is disabled, a message for it is queued rather than processed; if the method is later reenabled, the queued messages are processed.

Let the daemon itself appear as an object with a method

```
void sendMessage (Message&);
```

which is accessible by all the application objects on the node and is used to send messages to other objects. (An object may also have its own private message-sending methods that require that the argument be of some specific class derived from Message and that call sendMessage.)

While the above technique for sending and receiving messages makes the daemon application-independent, it may not be very convenient for the writer of the application classes, because s/he has to include logic to test the message type and call the proper method for the type. It is possible to give the daemon sufficient information to enable it to choose the proper method to invoke for each received message and still leave it application-independent. Two different ways of doing so are described briefly below.

The first way is for an object representing the receiving entity to be accessible to the sender. The entity's class can be given two different constructors, one for the "real thing" (the message receiver) and another for the sender's representative. The object implements methods

```
void sendMessage (const ParticularMessageType&);
 void receiveMessage (ParticularMessageType&);
```

for each particular message type it accepts. The sender invokes a method of the first form, which puts into the message an index into a method table maintained by the receiving version of the entity. The daemon on the receiving node uses the index to select and invoke the proper method of the second form. In addition to having access to an object of the proper type, the sender would have to have the receiving entity's global identifier, a quantity passed to it at run time. Two approaches could be taken: there could either be one sender's object for each receiving object, or one sender's object for each class of receiving objects. In the first approach, the receiver's identifier would be contained in the object, and in the second approach, the sender would supply the receiver's identifier whenever it sent a messages. If the second approach is taken, the above *sendMessage* operation would be changed to

```
void sendMessage (EntityID, const ParticularMessageType&);
```

The second way is to use an intermediate structure as message destination and source. Such structures are commonly called ports or channels. Suppose a sending entity sends to a port rather than to another entity, and a receiving entity receives from a port. Ports would have their own global identifiers which would be used in place of entity identifiers. One might use one port for each message type accepted by an entity. The port would reside on the same node as the receiving entity, and the receiving entity would register its ports with the daemon on its node, providing as part of the registration a pointer to itself and to a method to be called when a message directed to the port arrives. To provide type safety for the sender, the sender could use an object representing the port. Thus a port would have methods

```
void sendMessage (const ParticularMessageType&);
 void receiveMessage (ParticularMessageType&);
```

The sender would invoke the first, and the daemon on the receiver's node would invoke the second. The similarity to first method is evident.

Note that neither of the above methods requires any special preprocessor or compiler but could be programmed in plain C++. The claim of no special compilers or preprocessors is also made for the COMPOSE system. The *Charm* and *Charm++* systems bear a resemblance to the system proposed here but require a special preprocessor and do not implement guarded methods. The authors of the COMPOSE system have chosen another way of implementing inter-entity communication. In their scheme, typed messages are sent to entities (objects), as in the first technique outlined above. However, the class of the receiving object is not known to the sender. That appears to imply that before a sender and receiver can communicate, a preliminary internode communication must be done, similar to the binding of a client to a server, to enable the sender to obtain some sort of method identifier to put in its messages to the receiver. In addition to requiring further logic, this scheme could cause significant overhead for certain classes of programs (e.g., those in which this binding interaction cannot be amortized over a large number of communications with the same entity).

The model described here will work for any parallel program that can be stated as a collection of processes that send messages to each other. Selectively disabling of methods can conveniently be implemented by means of guard functions that return boolean values. A guard function callable by the daemon can be associated with each message-processing method. (Omitting a guard is equivalent to providing a guard that always returns a true value.) The same selector used to locate the correct method for the message can be used to locate the correct guard function for the message. If the guard returns a false value, the daemon retains the message rather than passing it to the entity by invoking its message-processing method. Every time a method of the entity is invoked it could change the output of any of the guard functions, so the daemon persists in passing messages to the entity until they are exhausted or until the guards for all the messages present have returned a false value. After that, the daemon will not trouble the entity again until a new messages for it arrives.

Several different policies could be used by the daemon in delivering messages (these correspond to queuing policies in a system of processes with message queues). The simplest is just to deliver the messages in arrival order. Another option is to give different priorities to different objects, to different message types or to individual messages. (Higher-priority objects receive their messages before lower-priority objects, messages of higher-priority message types are delivered before messages of lower-priority types, and higher-priority messages are delivered before lower-priority messages.) Such priorities are especially useful in real-time applications, where it may be necessary to process certain messages as soon as they come in.

Some further measures are needed for real-time systems. In order to process a message as soon as it comes in, preemptive priority scheduling is needed. Since the entities have no stacks of their own, any interruption of their methods must use a system stack. To support preemptive priority scheduling, a stack for each priority level is required. Then the daemon becomes multithreaded, with one thread for each priority level.

In PDES, the messages must be delivered in timestamp order, which is easily done by letting the timestamp be the message priority (a lower value having a higher priority).

Lack of stacks for the individual objects makes ordinary time-slicing infeasible, but something equivalent to it can be provided if priorities are available. Without time-slicing, an object could dominate its node with a long-running method; depending on the application, such behavior could decrease the overall parallelism of the system. Then the application programmer would have to observe the discipline of keeping the methods short enough that they do not become bottlenecks in the information flow. However, an effect similar to time-slicing can be obtained if a method's (or object's) priority is lowered dynamically as its running time increases.

It is possible to let objects invoke methods of other objects on the same node without going through the daemon. If priorities are used, it would be necessary to disallow direct calls to message-processing methods, though, because they would cause the priority ordering to be violated. It may be simplest to disallow direct calls to message-processing objects (entities) altogether. It would still be possible to use ordinary (non-message-processing) objects for all the usual purposes.

A.2 WAYS OF IMPLEMENTING CONCURRENCY

There is a small palette of options to choose from in defining the executing entities used in discrete-event simulation. There are two independent sets of two choices to be made: threads or no threads; and proxies or ports.

If threads are used, the user can wait for events (messages or timeouts) anywhere in the code. If threads are not used, receipt of a message (of a particular type) or a timeout is identified with the invocation of a (particular) method. We will call the executing entities with threads *active entities* and the executing entities without threads *passive entities*. Without threads, more entities can be packed on a node, and it is easier to move an entity to another node; the cost of having no threads is constraint to a "callback" style of programming.

An important difference between the two models has to do with program decomposition. In the active model, the entity can call a subroutine that receives messages and may eventually return results. In the passive model, the entity must create another entity to receive those messages, and any results must be passed back via messages. (In both models, entities may call ordinary subroutines that receive no messages but may return results.) A discussion of program decomposition will follow.

We wish to avoid having to use any special preprocessor or translator in defining a simulation language. In addition, we would like to avoid performing an explicit client/server-style binding between message sender and message receiver before sending the first message from that sender to that receiver. (It requires more logic and hurts performance, particularly when entity lifetimes are short, as they would often be with delegation.) The sender, however, must include something in the message that allows the receiver to tell what message type it is. Two ways of doing this are *proxies* and *ports*.

First proxies: let each entity type be associated with two classes, one a "sender's version" and the other a "receiver's version". One object of the receiving class is created, and it is the "real thing". Any number of objects of the sending class may be created; they are mere shells or proxies. However, a proxy does contain the information needed to properly tag all messages so that the receiving node can identify them. (We assume that the sender learns the entity ID of the receiver at run time.)

Now ports: ports are objects whose whole purpose is to receive messages. They are first-class entities in the sense that they have global identifiers. Ports are properly owned by and reside with the entity that receives from them. Global entity IDs are not used, only global port IDs. The sender learns port IDs at run time, just as it would learn entity IDs in the proxy scheme, and sends a message to a particular port. A special requirement of PDES is that entities consume incoming messages in timestamp order. This is equivalent to implementing message priorities, where the lower timestamp is the higher priority.

The next question to address is what the four options might look like to the programmer. To answer it, consider an example entity in each of the four styles. We use a Maisie entity used as an example in a paper by Bagrodia (1994). In Maisie, the entity looks like:

```
entity server {mean}
  int mean;
{ message job {int dep;} j1;
  message idmsg { ename id; }
  ename nextid;
  wait until mtype(idmsg) nextid = msg.idmsg.id;
```

```
            for (;;)
               wait until mtype (job)
               { j1 = msg.job;
                 hold (expon(mean));
                 invoke nextid with job = j1;
               }
          }
```

Figure A.1: An example Maisie entity.

In the examples below, note that a global identifier consists of two parts, a node number and a unique id number on that node. We assume that entities do not migrate from one node to another.

A.2.1 Active Entities Using Ports

In the active model with ports, both the sender's node and the receiver's node have an object representing a given port (i.e., both have the same global identifier for the port). The node number in the port's global identifier is that of the receiver's node (the receiver "owns" the port). The sender (an active entity) sends a message to the receiver by invoking the port's *send* operation. The port's *send* operation calls upon the daemon on the sender's node to deliver the message. The daemon knows about all ports with receivers on its node (the receiver registers them with the daemon). The daemon on the sender's node derives the receiver's node from the port's global identifier and sends the message to that node, tagged with the port's unique id number on that node (also derived from its global identifier). The daemon on the receiving node delivers the message to the port by invoking the port's *send* operation. The message is enqueued in the port until the receiver (an active entity) asks for it by invoking the port's *receive* operation. The usual techniques of mutual exclusion are used to keep the receiver and the daemon from interfering with one another.

In general, an entity wants to receive a message from any one of some subset of its ports. Of those ports in the subset that have messages ready, the entity must choose the message with the highest priority. The entity can handle this for itself without involving the daemon, provided the port makes its priority accessible (its priority is the priority of the highest-priority message enqueued in the port). It is convenient to encapsulate the logic to perform these functions in methods that are part of every entity. The *_request* operation allows the entity to record the ports it can receive input from; if none of the ports is ready, the *_wait* operation calls the daemon's *Wait* function, which returns when there is a message in at least one of the requested ports. The daemon knows which entity to awaken because the entities register their ports with it. The *_wait* operation selects from the input ports the one with the highest-priority message and returns its id. The returned id is not used in the example below because the entity receives from only one port at a time. In a windowed PDES protocol, the daemon would enqueue all the window's messages before beginning to schedule the execution of the entities. The following is an example of an active entity using ports.

```
class GlobalId {
  public:
    int node;
    int number;                // unique on node
```

```
    GlobalId (int,int);
    GlobalId (void);
};
typedef int EntityId;          // unique on this node
class BasePort {
  public:
    BasePort (GlobalId id, EntityId owner);
    BasePort (EntityId owner);
    BasePort (GlobalId id);
    BasePort (void);
    void _register (EntityId owner);  // so daemon knows whom to awaken
    Boolean ready (void);             // true if message enqueued
    double priority (void);           // e.g., timestamp
    GlobalId id (void);
    EntityId owner (void);
  protected:
    GlobalId _id;
  private:
    EntityId _owner;
 };
template <class T>
class Port : public BasePort {
  public:
    Port (GlobalId portId, Entity owner);
    Port (EntityId owner);
    Port (GlobalId id);
    Port (void);
    void receive (T& msg);      // gets highest-priority enqueued msg
    void send (const T& msg);   // transmit or enqueue message
  private:
...
};
class BaseMessage {
  public:
    GlobalId destination;
    double priority;
};
class JobMsg : public BaseMessage { ... };
class IdMsg : public BaseMessage {
  public:
    GlobalId id;
    EntityId owner;
};
class Entity {
  public:
    Entity (EntityId id);
  protected:
    void _request (const BasePort&);  // add to input ports
    GlobalId _wait (void);            // wait for msg on input port
    EntityId _id;
...
};
class Server : public Entity {
    Server (EntityId id, int mean,
            Port<IdMsg>& idPort, Port<JobMsg>& jobPort) :
      Entity(id), _mean(mean), _idPort(idPort), _jobPort(jobPort)
    {
      idPort._register(id);
      jobPort._register(id);
      CreateThread (Server::_body);
```

```
    }
  private:
    int _mean;
    Port<IdMsg> _idPort;
    Port<JobMsg> _jobPort;
    GlobalId readyPort;

    void _body (void) {
      IdMsg next;
      JobMsg job;
      _request (_idPort);          // accept input from _idPort
      readyPort = _wait();         // wait for input
      _idPort.receive(next);       // retrieve message
                                   //  use msg contents to create port
      Port<JobMsg> nextPort(next.id, next.owner);
               for (;;) {
        _request (_jobPort);
        readyPort = _wait();
        _jobPort.receive(job);
        Hold (expon(mean));
        nextPort.send(job);        // send message to  nextPort
      }
    }
};
```

Figure A.2: An active entity using ports.

A.2.2 Passive Entities Using Ports

In the passive model with ports, it is also true that both the sender's node and the receiver's node have an object representing a given port and that the node number in the port's global identifier is that of the receiver's node. The sender sends a message to the receiver by invoking the port's *send* operation. The port's *send* operation calls on the daemon on the sender's node to deliver the message. As in the active model, the sender's daemon derives the receiver's node from the port's global identifier and sends the message to that node, tagged with the port's id number on that node. The receiver has prepared the port by calling its *callback* operation, telling it which method to call when a message arrives. However, before calling the port's *send* method, the daemon on the receiving node first calls the ports's *guard* method. If the guard returns a false value, the daemon retains the message and attempts to deliver it again after the entity has received some other message. When the daemon on the receiving node does call the port's *send* method, the port logic invokes the specified callback method to deliver the message to the entity.

When message priorities are used, the daemon must deliver to the entity the highest-priority message that has a true guard. The daemon can do this by starting with its highest-prioity message and working down. If we wish to allow the daemon to give each entity all its messages before moving on to the next entity, the entities must register the ports with the daemon as in the active model; we will assume that the *callback* operation takes care of the registration. The following is an example of a passive entity using ports.

```
typedef int Boolean;
typedef int EntityId;
class Entity {
  public:
```

```
    Entity (EntityId id);
    EntityId id (void);
  protected:
    EntityId _id;
...
};
class BaseMessage {
  public:
    GlobalId destination;
    double timestamp;
};
typedef Boolean (Entity::*Guard)(BaseMessage& msg);
template <class T>
class Port {
  public:
    Port (GlobalId id, EntityId owner);
    Port (EntityId owner);
    Port (GlobalId id);
    Port (void);
    void send (const T& msg);
    void callback (Entity *obj, void (Entity::*method)(T& msg),
                   Guard guard);
    Boolean guard (void);
private:
    GlobalId _id;
    EntityId _owner;
  ...
};
class JobMsg : public BaseMessage { ... };
class IdMsg : public BaseMessage {
  public:
    GlobalId id;
};
class Server : public Entity {
  public:
    Server (EntityId id, int mean, Port<IdMsg>& idPort,
            Port<JobMsg>& jobPort, Port<BaseMessage>& timerPort) :
      Entity(id), _mean(mean), _idPort(idPort), _jobPort(jobPort),
      _timerPort(timerPort), _nextPort(NULL), _processingJob(FALSE)
     {
      _idPort.callback ((Entity *)this,
            (void (Entity::*)(IdMsg&))Server::_receiveId,
            (Guard) Server::_idGuard);
      _jobPort.callback ((Entity *)this,
             (void (Entity::*)(JobMsg&))Server::_receiveJob,
             (Guard)Server::_jobGuard);
      _timerPort.callback ((Entity *)this,
             (void (Entity::*)(BaseMessage&))Server::_receiveTimeout,
             (Guard)Server::_timeoutGuard);
    }
  private:
    int _mean;
    Port<BaseMessage> _timerPort;
    Port<IdMsg> _idPort;
    Port<JobMsg> _jobPort, *_nextPort;
    JobMsg _job;
    Boolean _processingJob;

    void _receiveId (IdMsg& msg) {
      _nextPort = new Port<JobMsg>(msg.id); }
```

```
    Boolean _idGuard (void) { return (_nextPort == NULL); }
    void _receiveJob (JobMsg& job) {
      _processingJob = TRUE;
      _job = job;
      Timer (expon(mean));         // requests timeout call
    }
    Boolean _jobGuard (void) {
      return (_nextPort != NULL && !_processingJob);
    }
    void _receiveTimeout (BaseMessage& ignored) {
      _processingJob = FALSE;
      _nextPort.send (_job);
    }
    Boolean _timeoutGuard (void) { return (_processingJob); }
};
```

Figure A.3: Passive entity using ports.

A.2.3 ACTIVE ENTITIES USING PROXIES

In the active model with proxies, the sender (an active entity) has access to a proxy object for the receiver. Every entity has a unique global identifier, and the receiving entity's global identifier is known to the proxy. The sender invokes the *sendMessage* in the proxy to send the message. The *sendMessage* operation corresponding to the type of message sent is automatically selected. The proxy passes the message to the sending node's daemon, tagged with the entity id and the type number corresponding to the message type, and the daemon transmits the message to the receiving node. A daemon is assumed to know about all entities on its node. The daemon on the receiving node enqueues the message and delivers it to the receiving entity when that entity requests it.

When message priorities are used, the entity must receive the highest-priority message that is ready among those message types in which it is currently interested. Since the daemon must choose the message to deliver, the daemon must know which types are candidates for the entity. The entity registers its interests with the daemon with the *Request* operation and then waits for arrival of a message with the *Wait* operation, which returns the type of the ready message. (In the example below, only one message type is wanted at a time, so the returned message type is not used.) The entity may then retrieve the message with the *ReceiveMessage* operation. The following is an example of an active entity using proxies.

```
class Entity {
  protected:
    GlobalId _id;
    Entity (GlobalId id) : _id(id) {
      Register(this, id);    // let daemon know about this entity
    }
};
class BaseMessage {
  public:
    GlobalId destination;
    double timestamp;
};
class JobMsg : public BaseMessage { ... };
class IdMsg : public BaseMessage {
```

```
  public:
    GlobalId id;
};
class ServerProxy {
  public:
    ServerProxy (GlobalId);
    void sendMessage (const IdMsg&);
    void sendMessage (const JobMsg&);
    void sendMessage (const BaseMessage&);
};
class Server : public Entity {
  public:
    Server (GlobalId id, int mean) :
      Entity(id), _mean(mean),
      _processingJob(FALSE), _nextServer(NULL) {    }
  private:
    int _mean;
    JobMsg _job;
    Boolean _processingJob;
    ServerProxy *_nextServer;
    void _body (void) {
      IdMsg msg;
      JobMsg job;
      int messageType;

      Request(_id,0);               // request type 0 message
      messageType = Wait();         // wait for msg of that type
      ReceiveMessage(_id,0,msg);    // and retrieve it
      _nextServer = new ServerProxy(msg.id);
               for (;;) {
        Request(_id,1);
        messageType = Wait();
        ReceiveMessage (_id, 1, job);
        Timer(expon(mean), 2);      // request timeout message
        Request(id,2);
        messageType = Wait();
        ReceiveMessage (_id, 2, NULL);
        _nextServer->sendMessage(job);
      }
    }
```

Figure A.4: Active entity using proxies.

A.2.4 PASSIVE ENTITIES USING PROXIES

As in the active model with proxies, it is true that the sender has access to a proxy object for the receiver, that each entity has a unique global identifier, and that the receiving entity's global identifier is known to the proxy. As in the active case, the sender invokes the proxy's *sendMessage* method to send a message to the receiver, and the *sendMessage* operation corresponding to the message type is automatically selected. The proxy passes the message to the sending node's daemon, tagged with the entity id and the type number corresponding to the message type, and the daemon transmits it to the receiving node. The daemon on the receiving node locates the receiving entity, using the global identifier in the message, and delivers the message by invoking the method corresponding to the message type in the message (the message type is the index of the method in the entity's method table). But before invoking the delivery method, the daemon invokes the guard indicated by the index; if the guard returns a false value, the daemon retains the message and does not try to deliver it again until after some other message has been successfully delivered to the entity.

If message priorities are used, it is necessary for the daemon to deliver the highest-priority message of those that the entity might receive. With proxies, the messages contain the entity id, so it is a simple matter for the daemon to select the highest-priority one to deliver. The following is an example of a passive entity using proxies.

```
class Entity {
  protected:
    GlobalId _id;
    Entity (GlobalId id) : _id(id) {
      Register(this, id);    // let daemon know about this entity
    }
};
class BaseMessage {
  public:
    GlobalId destination;
    double timestamp;
    int type;
};
typedef void (Entity::*Method)(BaseMessage& msg);
typedef Boolean (Entity::*Guard)(void);
class JobMsg : public BaseMessage {
  public:
    GlobalId destination;
    double timestamp;
};
class IdMsg : public BaseMessage {
  public:
    GlobalId id;
};
class ServerProxy {
  public:
    ServerProxy (GlobalId);
    void sendMessage (const IdMsg&);
    void sendMessage (const JobMsg&);
    void sendMessage (const BaseMessage&);
};
class Server : public Entity # 1 {
  public:
    Server (GlobalId id, int mean) :
      Entity(id), _mean(mean), _processingJob(FALSE), _nextServer(NULL)
    {
      method[0] = (Method)Server::_receiveId;
      guard[0]  = (Guard) Server::_idGuard;
      method[1] = (Method)Server::_receiveJob;
      guard[1]  = (Guard) Server::_jobGuard;
      method[2] = (Method)Server::_receiveTimeout;
      guard[2]  = (Guard) Server::_timeoutGuard;
    }
    Method method [3];
    Guard  guard [3];
  private:
    int _mean;
    JobMsg _job;
    Boolean _processingJob;
    ServerProxy *_nextServer;
```

```
    void _receiveId (IdMsg& msg) {
      _nextServer = new ServerProxy (msg.id);
    }
    Boolean _idGuard (void) { return (_nextServer == NULL); }
    void _receiveJob (JobMsg& job) {
      _processingJob = TRUE;
      _job = job;
      Timer (expon(mean));         // requests timeout call
    }
    Boolean _jobGuard (void) {
      return (_nextServer != NULL && !_processingJob);
    }
    void _receiveTimeout (BaseMessage&) {
      _processingJob = FALSE;
      _nextServer.sendMessage (_job);
    }
    Boolean _timeoutGuard (void) {
      return (_processingJob);
    }
};
```

Figure A.5: Passive entity using proxies.

A.2.5 Program Decomposition

Program decomposition is fundamental to software development. It is the means of breaking a complex entity up into simpler pieces. Decomposition can be sequential or parallel. Sequential decomposition just amounts to calling a subroutine. Parallel decomposition is accomplished by creating entities. The possible combinations of active/passive, ports/proxies, and sequential/parallel generate eight cases, which are discussed separately below.

1) <u>Active/Ports/Sequential</u>: The entity can pass ports as arguments to a subroutine, which can use them without any special measures.

2) <u>Active/Ports/Parallel</u>: The entity can pass ports to an entity it creates, either as arguments to its constructor or in messages. The created entity must register the ports to itself, so the daemon will know what process to schedule when a message arrives. The creator must register the ports back to itself when the created entity terminates.

3) <u>Active/Proxies/Sequential</u>: A called subroutine can receive messages from the daemon as well as the caller; the entity's id remains the same.

4) <u>Active/Proxies/Parallel</u>: The sender sends to a specific entity id (via the proxy), and the entity id of a created entity is different from that of the creator. Therefore the creator entity must ask its daemon to redirect messages sent to its id instead to the created entity's id. Before it terminates, the created entity must first redirect messages back to the parent. The redirection should be done by message type, so that the creator can pass the responsibility for different types to different created entities.

5) Passive/Ports/Sequential: Something akin to sequential decomposition can be obtained by setting a port's callback method to point to a different routine.

6) Passive/Ports/Parallel: The parent entity can create a child entity and pass it ports via constructor arguments or in a message. The child then sets the callbacks. Before it terminates, the child informs the parent in a message that it is passing the ports back. The parent then sets the callbacks back to its own methods.

7) Passive/Proxies/Sequential: Same as 5).

8) Passive/Proxies/Parallel: After it creates the child, the creator entity must ask its daemon to redirect messages to it. The created entity must direct them back to its creator before it terminates. The redirection should be done by message type, so that the creator can pass the responsibility for different types to different created entities.

A.3 AN INTERPRETATION OF CHANDY-SHERMAN SPACE-TIME SIMULATION

A completely new way of looking at discrete-event simulation was presented in the brief paper "Space-Time and Simulation", by Chandy and Sherman (1989). Though the paper is sometimes cited, I know of no simulation system that embodies the principles outlined in it. Perhaps the reason is that the paper is not easy to understand. It is condensed, even delphic. This write-up attempts to interpret the paper and carry it to a point at which a practical implementation of its techniques can be made.

An analogy can be drawn between carrying out a parallel discrete-event simulation and solving a boundary-value problem by the relaxation (successive approximation) method. In a boundary-value problem, it is desired to compute values which satisfy a certain criterion (perhaps they are a solution of a given equation) over a set of points, given information about the values on part of the point set (the boundary).

A PDES simulates a set of physical processes. It is the object of the PDES to compute a (correct) simulation-time history of the outputs of the physical processes from the starting simulation time (0) to the simulation horizon (H), given certain assumptions about the initial states of the physical processes. The PDES simulates the physical processes by using a set of logical processes. In the simplest case, there is a one-to-one correspondence between physical and logical processes. The outputs of the physical processes are deducible from the outputs of the logical processes (in the simplest case identical to them).

Take the logical processes in the PDES to be analogous to the points in the boundary-value problem. The values we wish to compute at these "points" are the complete (correct) simulated time histories of the logical processes over the interval [0, H]. These histories consist of a sequence of messages with timestamps in increasing simulation-time order. A partial sequence can be considered to be an approximation of the desired value. Similarly, a (partial) sequence in which the trailing messages are incorrect can be considered to be an approximation of the desired value. (The individual messages are somewhat like the decimal digits of a numeric value. As the simulation runs, the approximation of the desired value becomes better and better.)

Instead of making assumptions about the initial states of the physical processes, we can specify the initial state in a PDES by means of a set of initializing messages sent to the logical processes. The time-history of the sources of initial messages can be taken to be analogous to the boundary values in the boundary-value problem. In general, the "physical process space" can be covered by nonoverlapping regions in any fashion, with each region representing a logical process. The covering can even change with time. The "points" are still taken to be the logical processes; that is, the relaxation is carried out in "logical process space". From now on, when we say "process", we will mean "logical process".

In the relaxation method, the current value at a point (the output at that point) is computed using the past values at neighboring points as input. In a synchronous algorithm, a point cannot perform the n^{th} iterative computation of its value until all the neighbors from which it inputs have performed their $(n - 1)^{\text{st}}$ iteration. (If a point requires up to k past values of its neighbors in computing its current value, the algorithm is called "k^{th} order in time".) In a chaotic algorithm, the current value at a point may be computed at any time, using the currently available values of its neighbors (the currently available k past values if the algorithm is k^{th} order in time).

The situation is analogous in a PDES. A process computes its value in the form of its output messages, which are used as input by other processes (its "neighbors") in order to compute their values. The points may exchange information besides the current approximation of the final value of interest; said another way, it is possible that only part of the output produced at each point may be used in the final ("external") output of the program. Likewise, in a PDES, some subset of the output messages may be sent outside the simulation to serve as the simulation's "external output". An output message is either used as input by another process or is part of the external output.

Let the simulated time variable be x and the real time variable be t. The principle of causality says that the computation of an output message with timestamp x can depend only on input messages with timestamps less than x. This corresponds to the fact that in a relaxation algorithm, the current value is computed from the past values of the neighbors.

Consider a graph with a node representing each point (process) and a directed arc connecting two points if the first point provides input for the second. If the graph is cyclic, we say that there is "feedback" in the system. In general, non-trivial systems treated by DES have feedback (an acyclic system could be treated with a systolic computation).

One result of considering discrete-event simulations in this fashion is that the distinction between real-time and non-real-time simulation is removed. A (non-trivial) discrete-event simulation corresponds to a (possible) real-time simulation of a system with feedback. In the real-time simulation, the messages which are known to be correct are output as they are produced, under the constraint that when they are output, the real time t must be within a certain tolerance of the simulated time x. A time-stepped real-time simulation would correspond to a synchronous relaxation algorithm. An asynchronous real-time algorithm would correspond to a chaotic relaxation algorithm.

A.3.1 Description of a Chaotic PDES Algorithm

Suppose the causality principle holds. Suppose each process continually goes through a cycle of reading input values and consuming them to produce output values. For the purposes of this section, suppose that at each cycle, the process outputs all its messages from time 0 on at each cycle. For simplicity, suppose that simulated time is an integer variable.

At each cycle, let the process compare its output sequence for each of its message destinations with the output sequence it developed for that destination on its previous cycle and note the greatest simulated time through which the two sequences are the same. (If the first differing message has timestamp x, the two sequences are the same through simulated time (x – 1).) Let the process tag its current output for that destination with this time.

Messages that have been output but not yet read and used to produce output are called "outstanding" messages. We claim that the system has converged through the lowest tag time in any outstanding message.

The claim follows from causality. Suppose a process's input on the current cycle is the same up to simulated time x and different from that point on. Then any difference in the process's output on this cycle from its output on the previous cycle must consist of messages with timestamp greater than x. Let the minimum of the message tag times be x_c. It follows from the definition of the tag time that there is no outstanding message that can cause a process to produce on its next cycle any output with timestamp less than x_c that is different from what it produced on the previous cycle. Therefore the system has converged through simulated time x_c. As a sidelight, note that if the physical system being simulated has the property that it consists of causal physical processes, i.e., processes whose state at time $t1$ depends on their state at times $t < t1$ and the external influences (messages) at times $t < t1$, it ought to be possible to construct a causal simulation of the system.

A.3.2 Conservative and Optimistic Simulation

If the most recently computed convergence time x_c is known to a process before it reads its inputs for a cycle, it can regard all inputs having timestamps no greater than x_c as being provably correct. Any outputs produced using only these inputs are likewise provably correct. (Each process handles its input messages in timestamp order. If the first input message that is not provably correct has timestamp x, then the process can produce no provably correct output with timestamp greater than $x - 1$.)

If a process produces only provably correct output, it is behaving conservatively. If it also produces output that is not provably correct, it is behaving optimistically. A process may alternate between conservative and optimistic behavior, behaving conservatively whenever it can and optimistically whenever it would otherwise be idle. Suppose a process has read its input messages for a cycle and used them to produce all the provably correct output it can. If at that point its input for a new cycle has arrived, it can discard the remaining (unprocessed) input for this cycle and begin the next cycle, having produced only provably correct output for this cycle. If, on the other hand, it has no new input, it can behave optimistically and process the rest of its input from this cycle to produce output that is not provably correct.

A.3.3 Another Way of Looking at the Relaxation/PDES Correspondence

To pursue another analogy, consider a relaxation algorithm that develops its output value as the coefficients of a expansion in terms of some basis vectors. Suppose that the result is a vector result, and that each process develops one component of this vector. The inputs to each process are the approximate values of (some of) the other components.

The coefficients correspond to messages and the indices of the coefficients to the message timestamps. (A process will not necessarily output a value for every index.) A conservative PDES algorithm corresponds to an algorithm that outputs only coefficients that are known to be correct. An optimistic PDES algorithm corresponds to an algorithm that outputs coefficients that are only approximately correct.

Let the j^{th} coefficient of the i^{th} vector component be c^i_j. Then the causality principle would take the form: the value of a coefficient c^i_j depends only on the coefficients $\{c^m_n: m \neq i, n < j\}$. If we wish to obviate (conceptually, at least) the need for any process to remember its state, we could say that c^i_j depends only on $\{c^m_n: n < j\}$. Then the initial state would be specified by giving some initial coefficient values.

A.3.4 Discovery of the Convergence Time

Suppose each node (processor) on a parallel system runs its local logical processes using the object-oriented model. In this model, each process corresponds to an object (or entity), and all messages sent from one entity to another are routed through a daemon which runs on each processing node of the computing system. Delivery of a message to an entity on the node corresponds to invocation of one of the entity's methods by the daemon. Thus the daemon knows whenever any input message is consumed by any entity on the node.

In order to compute the convergence time (call it x_c), it is necessary to account for all outstanding messages. If a message's source and destination are on the same node, the daemon on that node will be able to account for the message. If the source and destination are on different nodes, the message can be accounted for only after it has been received by the daemon at the destination.

Each node's daemon can develop an estimate of x_c as the minimum of the tag times in all messages received but not yet consumed on the node. The daemons on all the nodes can from time to time perform a reduction to discover the minimum of all the estimates. This minimum is the new convergence (converged-to) time. A standard reduction algorithm is not adequate, since it does not account for messages in transit. Nicol's noncommittal barrier algorithm (1995) is suitable, however, since it delays completion of the reduction until all messages have been accounted for.

Note that the scheme would also work if each node, instead of using the OO model, did a sequential simulation using either the process model or the event-list model. In all these cases, a controlling program on the node would be in a position to have the required knowledge about messages sent, received and consumed.

A.3.5 High-level Statement of an Implementation of the Algorithm

In any implementation, the processes would not output their entire message sequences from time 0 at each cycle. It is only necessary for them to output the part of the sequence that is different from the previous cycle's sequence. Likewise they would retain internal state so that they would not have to compute the entire sequence anew at each cycle. They would, however, have to keep two representations of that state, one "correct" state representation, which is advanced in response to the advancement of the convergence time, and one state representation which is developed in the course of producing output that is not provably correct (optimistic output). Assume, for simplicity, that the current estimate of the convergence time is available to all processes on a node in a global variable x_c. Assume also that the node's daemon does not update x_c after a process has read its input messages but before the process has produced the output (if any) it is going to produce in response to them.

```
do

    read all input messages now available to this process

    process input messages in timestamp order
    until run out of messages which are marked
    "proven" or which have timestamps no greater
    than x_c, producing (but not yet sending) output
    messages marked "proven" and updating the state
    of this process.

    if no further unread input has become available –>

        make a copy of the state of this process and
        process the rest of the already-read input
        to produce (but not yet send) further output
```

```
        messages, updating the copy of the state.
        (this copy can then be discarded).

    fi

    for each destination represented in
        the output just produced or in the
        output produced on the previous cycle —>

         discover the greatest time through
         which the output produced on this
         cycle is the same as the output
         produced on the previous cycle

    rof
    tagtime := the least of the times developed above

    for each destination represented
        in the output just produced  —>

         send the part of the output that has
         timestamps greater than tagtime
         and include tagtime in the messages

    rof

od
```

Figure A.6: A high-level statement of an implementation of the algorithm.

A.4 PARALLEL CONSERVATIVE SIMULATION TIMING DIAGRAMS

During the course of IDES research, a great deal of time was spent studying conservative simulation protocols, and their timing nuances. The following diagrams highlight two of these subtle aspects.

For each, the timing diagram legend in Figure A.7 applies. The horizontal axis represents simulation time for each of the entities. Activities can be interruptible or non-interruptible. ΔTa is the minimum activity length of time. ΔTc is the commitment time. If an interruptible activity with a scheduled length of T reaches time T – ΔTc, it can no longer be interrupted. Each entity offers a window bid time. The entity guarantees it will not produce a message for another entity with a time stamp less than its window bid, assuming it does not receive any new messages from other entities.

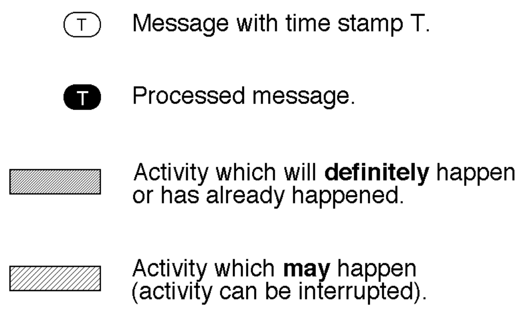

Figure A.7: Timing diagram legend.

Figure A.8 demonstrates the issues associated with non-interruptible activities. The first drawing is initial state. All entities are synchronized at time T0. Entity A has an unprocessed message (received in an earlier window) which is to take place at time T1.
Minimum window bid is T1 + ΔTa.

The second drawing shows state after processing each entity up to the window edge T1 + ΔTa (synchronization point). During this window, Entity A processed its message at time T1, learning that the activity is to last until T2 and that at that time (T2), a message should be received by Entity B. As soon as this information was known, Entity A "pre-sent" the message to Entity B. At the synchronization point, Entity B has an unprocessed message with time stamp T2, so it calculates its window bid as T2 + ΔTa. The minimum window bid is T2 + ΔTa.

The final drawing shows state after processing each entity up to the window edge T2 + ΔTa. During this window, Entity B processed its message at time T2, learning that the activity is to last until T3 and that at that time (T3), a message should be received by Entity C. As soon as this information was known, Entity B "pre-sent" the message to Entity C. At the synchronization point, Entity C has an unprocessed message with time stamp T3, so it calculates its window bid as T3 + ΔTa.

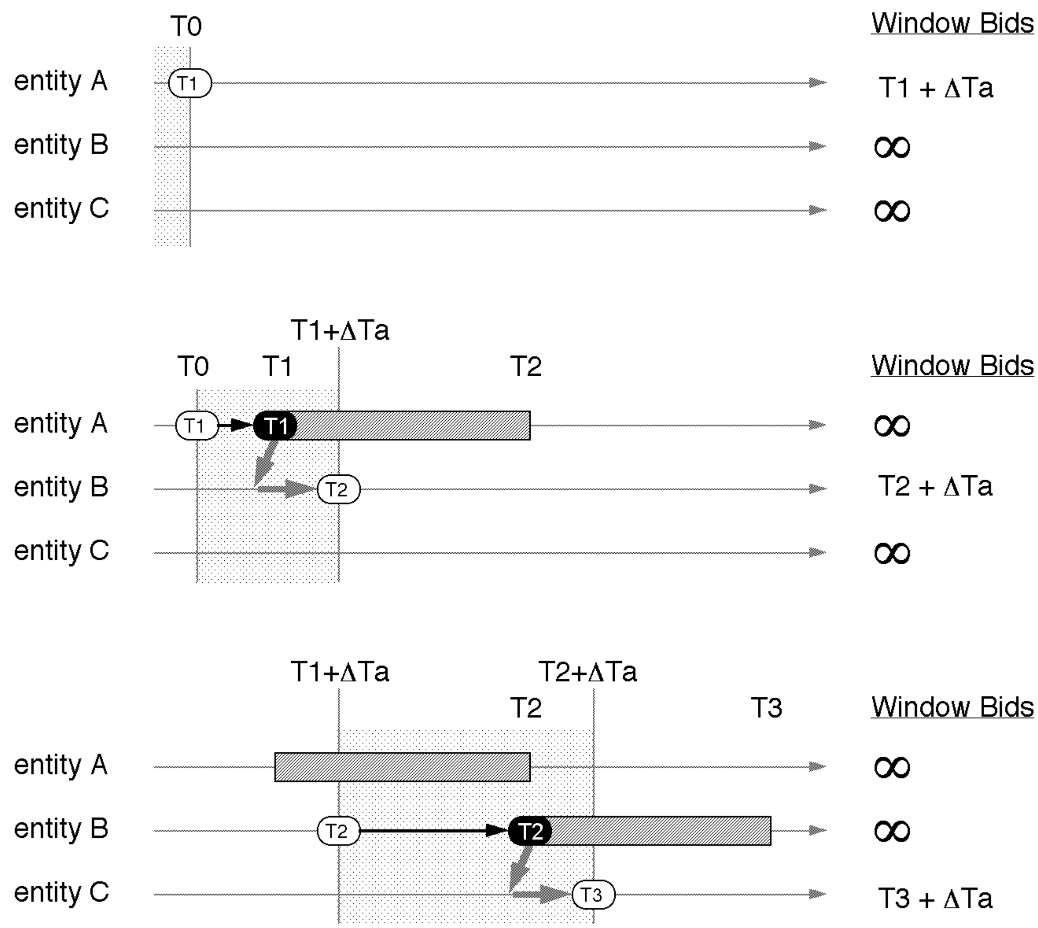

Figure A.8: Non-interruptible activities.

Figure A.9 documents interruptible activities. The first drawing is initial state. All entities are synchronized at time T1. Entity A has an unprocessed message that is to take place at time T1. The minimum window bid is T1 + ΔTa.

The second drawing shows state after processing each entity up to the window edge T1 + ΔTa. During this window, Entity A processed its message at time T1, learning that the activity is to last until T2 and that at that time (T2), a message should be received by Entity B. Since the activity is interruptible, Entity A cannot be certain that the message to Entity B should really be sent, so it cannot "pre-send" the message yet. Based on the scheduled activity time, Entity A bids a window time of T2. The minimum window bid is T2.

The third drawing shows state after processing each entity up to the window edge T2. During this window, at time T2 – ΔTc, Entity A was committed to finishing its activity (could no longer be interrupted). At that time, Entity A could safely "pre-send" its message to Entity B with time stamp T2. At the synchronization point, Entity B has an unprocessed message with time stamp T2, so it calculates its window bid as T2 + ΔTa. The minimum window bid is T2 + ΔTa.

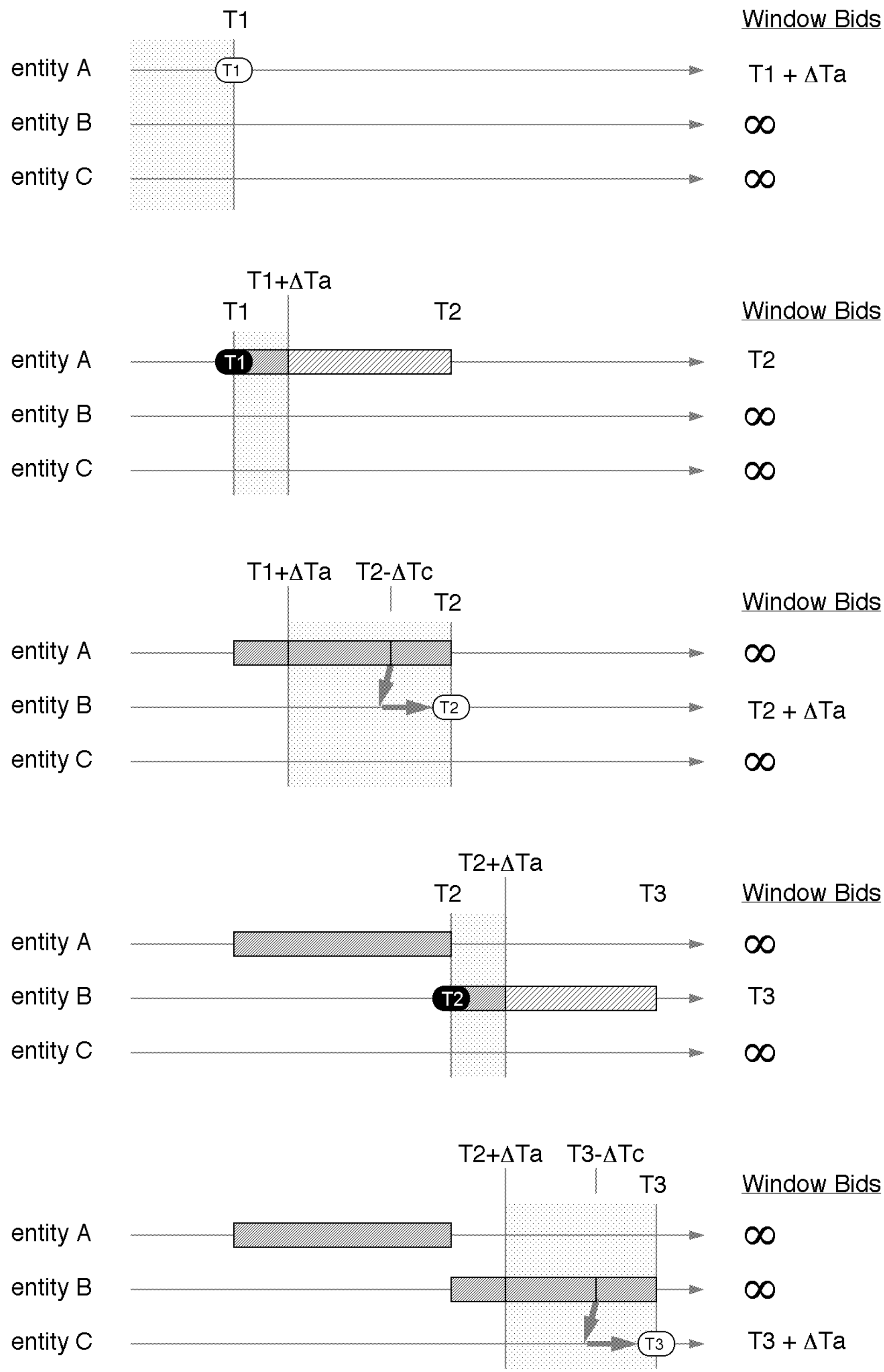

Figure A.9: Interruptible Activities.

The fourth drawing shows state after processing each entity up to the window edge T2 + ΔTa. During this window, Entity B processed its message at time T2, learning that the activity is to last until T3 and that at that time (T3), a message should be received by Entity C. Since the activity is interruptible, Entity B cannot be certain that the message to Entity C should really be sent, so it cannot "pre-send" the message yet. Based on the scheduled activity time, Entity B bids a window time of T3. The minimum window bid is T3.

The final drawing shows state after processing each entity up to the window edge T3. During this window, at time T3 – ΔTc, Entity B was committed to finishing its activity (could no longer be interrupted). At that time, Entity B could safely "pre-send" its message to Entity C with time stamp T3. At the synchronization point, Entity C has an unprocessed message with time stamp T3, so it calculates its window bid as T3 + ΔTa.

It should be noted that after the initial state, interruptible activities required two synchronization windows per message sent, while non-interruptible activities required only one synchronization window per message.

APPENDIX B: PREEMPTIVE MIN-REDUCTION ALGORITHM PROOF

In this appendix we prove that if we use the preemptive min-reduction algorithm to find the end of a BTB window on the parallel communication architecture, the execution time is no more than $2m\lceil \log P \rceil$ larger than the time needed if communication is instantaneous. The result follows from an analysis of how values propagate through the communication tree used to implement the min-reduction. In particular, we look at the effects of communicating the value that ultimately becomes the global event horizon.

For simplicity of exposition we take P to be a power-of-two. When a processor enters a min-reduction, it synchronizes pairwise with a sequence of $\log P$ processors. At each synchronization it sends a working minimum (initially its own value) to its partner, and does not proceed until receiving its partner's working minimum. Both processors keep the minimum of these two values as the working minimum, and proceed to the next stage. A processor's partner in the first stage is obtained by inverting bit 0 (the least significant bit) of the processor's id; its partner in the second stage is obtained by inverting bit 1, and so on.

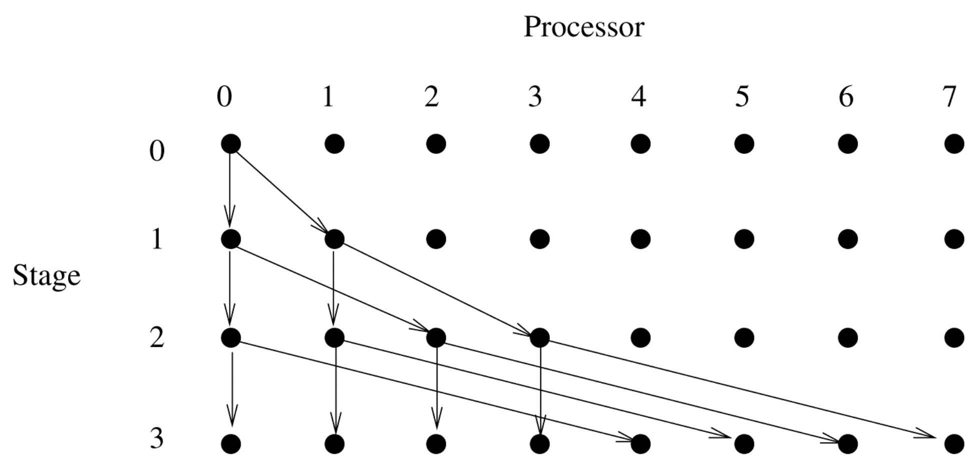

Figure B.1: Broadcast tree distributing the global event horizon when the critical transition occurs at processor 0, for P = 8.

Consider any given window, and without loss of generality, assume that processor 0 generates its critical transition. We can construct a tree illustrating how the global minimum defined by processor 0 spreads out through the processors stage-by-stage, illustrated in Figure B.1 for the case of P = 8. We see that every processor can be thought of as being "attached" to this tree at some stage, by receipt of the minimum value from its parent in the tree.

We assume that at the point a processor is first attached to the broadcast tree, it either stops because of the message sent by its parent, or continues until its time-of-next-event is as large as the value sent by that parent. In reality the child may have stopped earlier; the time required to complete the reduction cannot be lessened by assuming that the parent must explicitly wait for the child to send a synchronization message in response to the parent sent message. Under this assumption we construct a new graph that reflects message dependencies in the construction of the broadcast tree. This graph uses an ordinary node to represent a processor that is first attached to the tree; later participation by an attached processor is depicted as a double-circle. Nodes are labeled by the processors they represent, arcs depict message dependencies. The graph corresponding to a critical transition at processor 0, for P = 8, is shown in Figure B.2.

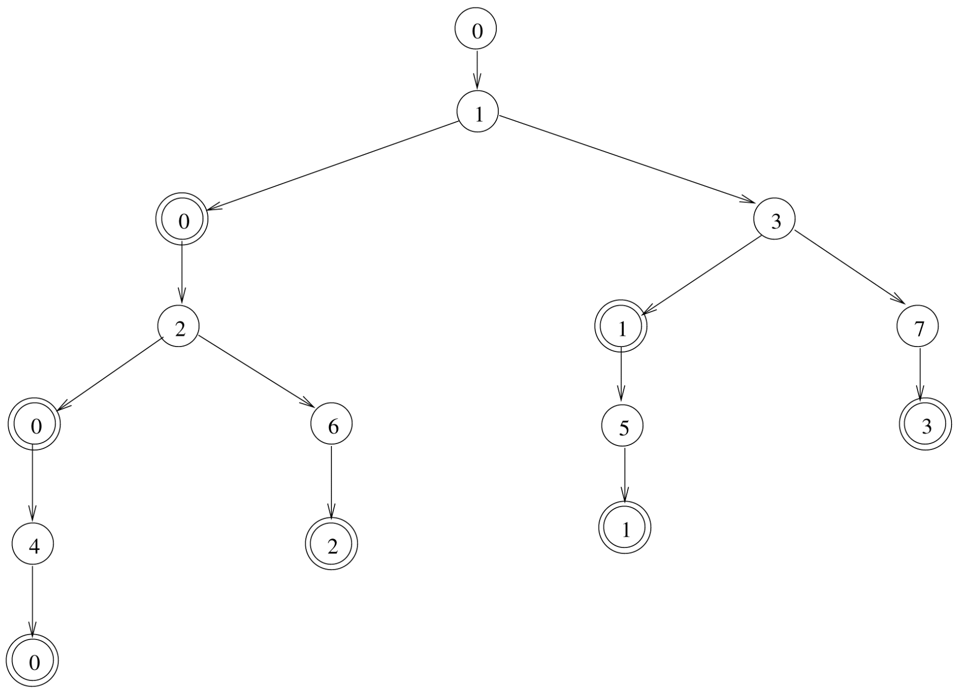

Figure B.2: Worst-case communication dependency graph for reduction-tree when the critical transition occurs at processor 0, for P = 8.

The time to complete the reduction is given as the longest path from source node to any leaf, using a specialized path measure. Every processor has exactly one ordinary node in the graph, which we weight by the processor's stopping time in the ideal case (i.e., the stopping time given by the analytic model). Edges are weighted by the communication delay between source and target processors, i.e., either 0 or m depending on whether the processors reside in the same machine. The length of the path from root "through" a specified node is defined recursively. The length of the path through the root is the weight given to the root node. The length of the path through a double-circle node is the length of the path through its parent, plus the weight on the edge between them. The length of the path through an ordinary node is the maximum of (i) its node weight, and (ii) the length of the path through its parent, plus the weight of the edge between them. We claim that the length of the path through a node is an upper bound on the time at which the synchronization represented by that node. We prove this by induction on the tree structure. The base case is satisfied: the root node represents the processor defining the critical transition; it cannot be stopped by receipt of any synchronization message, because its local event horizon is least. Its stopping time is the same with communication costs as it is without. For the induction hypothesis we presume that the assertion holds true for all nodes in a subtree that includes the root. Choose a node d that is not in the subtree, but whose parent is. By the induction hypothesis, the arrival time of the message from the parent to d is bounded from above by the length of the path through its parent plus the communication cost. If the node is a double-circle node, this message arrival frees the processor to engage in synchronization at the next stage, and so the path length through d is indeed an upper bound on the time when the synchronization represented by the node is completed. If d is an ordinary node, the arrival of the parent's message at time t may stop the processor d represents; if so it must be at a time at least as large as the weight given to the node, which is to say that if the node weight is less than t then the time at which the node sends its synchronization message to its parent is no greater than t, and the path length through d is exactly t. If d's node weight is larger than t, then in the system with communication delays, the associated processor will stop at the time given in its node weight, and not before. In this case the appropriate length of path through d is just the node weight. Consideration of these two cases shows that the path length through d is an upper bound on the completion time of the synchronization represented by d, completing the induction.

The longest possible path from source to any leaf is obtained when the source node is weighted with the largest stopping time among all processors, i.e., the overall window termination time in the ideal case. Then, if every communication edge from root to leaf was weighted by m, the path length is the ideal window termination time, plus $2m \log P$.

APPENDIX C: COMPLEXITY OF SOLVING MODEL BEHAVIOR EQUATIONS

In this section we sketch the computational complexity of solving the various equations describing our model's behavior. The equations may be solved numerically if we discretize the state-space, and apply discrete summations in place of integrals. The state-space is effectively finite, as we do not need to consider any states that represent time-stamps beyond the end of the simulation time. As a first approximation we replace the two-dimensional continuous state-space of each step with an $N \times N$ grid of discrete points, equispaced, and consider the computational complexity of solving the equations on that set of spaces.

Equations (1) and (2) are essentially convolutions in one variable. The direct approach entails $O(N)$ operations for each discretized state (s, r) (note that the inner integral in the first term of (2) may be pre-computed once and used for the solution of every state whose receive-time component is r). There being $O(N^2)$ states, the complexity of computing all state values at one step is $O(N^3)$. However, convolutions can be computed more efficiently using Fast Fourier Transforms. Some trickery is needed to express the convolution integrals "classically" as covering $-\infty$ to ∞, this is accomplished recognizing that the time-increment and receive-time increment density functions are zero with negative arguments. In this way, given fixed r, the $O(N)$ values of $d_i^{(k)}(s, r)$ for all s are computed in $O(N \log N)$ time. This lowers the per-step computational complexity to $O(N^2 \log N)$.

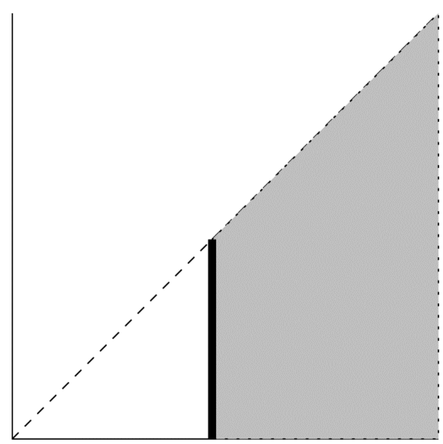

Figure C.1: Area of domain integrated to obtain $L_j(r')$, and the difference between $L_j(r' + \delta)$ and $L_j(r')$

The rest of the model involves conditioning on r', which means that for each variable $O(N)$ different values will need to be computed, one for each distinct discretized value of r'. At first glance it would seem that some values, (i.e. $L_j(r')$) require $O(N^2)$ time for each value of r', because of the double integration. However, this is not the case, because the value of a variable for one value of r' is computable with only $O(N)$ work from the value of the variable with the next closest value of r'. This is most easily seen using a diagram; consider Figure C.1. The shaded area illustrates the region of state-space over which $d_j^{(k)}$ is integrated to compute $L_j(r')$, for some value of r'. The single vertical line at the left edge of this space illustrates the region of the state-space that is not included in the value of $L_j(r' + \delta)$; we obtain $L_j(r' + \delta)$ by subtracting the contribution of this line from $L_j(r')$; computing the line's contribution costs only $O(N)$ time. The same sort of method works computing the numerator and denominator of $\gamma_j^{(k)}(r')$ (equation (4)), and for computing the areas illustrated in Figure C.1. The cost of computing all r' dependent values at a given step for all processors is thus $O(PN^2)$, a cost that is dominated by the initial cost of computing the unconditional state probabilities.

The number of steps we solve for will depend on the behavior of the underlying distributions. We will need to solve for as many steps as it takes before the probability of reaching the step is sufficiently small to ignore. Let K be this number of steps; the complexity results are more cleanly stated if we assume $K \geq N$.

The costs identified so far show that if the state occupancy probability functions $f_j^{(k)}$ are known, the cost of computing the distribution of $G(i, n, r)$ is $O(PNK)$. Considering the cost of computing the distributions of variables T_j (equations (5)) and (6)), observe that for $y > 0$, $\Pr\{T_j(n, r) = y\} = \Pr\{T_j(n + 1, r) = y - 1\}$. Coupled with our previously noted ability to compute $S_{j,*}$ type distributions incrementally, this symmetry can be exploited to streamline the computation of the all T_j distributions, obtaining them all in $O(PNK)$ time. The symmetry can be exploited again when computing the distribution of the max term in $W(i, n, r')$ (equation (8)); $O(PNK)$ time again is needed to compute distributions of all variables $W(i, n, r')$. The distribution of W (equation (8)) is then computed directly in $O(PNK)$ time.

Computation of the distributions for $L_j(r')$ and $X_j(i, n, r')$ uses the same ideas. Computation of $E_j(i, n, r')$ type distributions involves rescaling $O(K)$ values for every value of n. The complexity of computing distributions for all variables $E_j(i, n, r')$ is thus $O(PNK^2)$. The expectations $U(i, n, r')$ (equation (10)) are all computed in time $O(PNK)$, as is the final expectation U (equation (11)).

The cost of computing all the state values is $O(PKN^2 \log N)$, whereas the dominant cost of computing performance measures is $O(PNK^2)$. This is in some sense almost as good as one could expect given that state point has four coordinates: processor, the two state components, and time; the cost will be at least linear in the product of the extent of those dimensions. Our complexity figures are close to that optimum.

DISTRIBUTION:

David M. Nicol
Department of Computer Science
6211 Sudikoff Laboratory
Dartmouth College
Hanover, New Hampshire 03755-3510

1	MS 9001	T. O. Hunter, 8000
		Attn: J. B. Wright, 2200
		J. F. Ney (A), 5200
		L. A. West, 8200
		W. J. McLean, 8300
		R. C. Wayne, 8400
		P. N. Smith, 8500
		P. E. Brewer, 8600
		T. M. Dyer, 8700
		L. A. Hiles, 8800
		D. L. Crawford, 8900
1	MS 0149	C. E. Meyers, 4000
1	MS 9004	M. E. John, 8100
1	MS 9201	L. D. Brandt, 8112
1	MS 9201	P. K. Falcone, 8114
1	MS 9201	M. E. Goldsby, 8114
10	MS 9201	M. M. Johnson, 8114
1	MS 9201	A. S. Yoshimura, 8112
1	MS 9214	L. M. Napolitano, 8130
3	MS 9018	Central Technical Files, 8940-2
4	MS 0899	Technical Library, 4916
1	MS 9021	Technical Communications Department, 8815/Technical Library, MS 0899, 4916
2	MS 9021	Technical Communications Department, 8815, for DOE/OSTI